BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2020 Bc. Martin Horák

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

SEMANTIC DESCRIPTION OF THE EMBEDDED DEVICE SCREEN

SÉMANTICKÝ POPIS OBRAZOVKY EMBEDDED ZAŘÍZENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Martin Horák**
AUTOR PRÁCE

SUPERVISOR — **Ing. Ilona Janáková, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study field **Cybernetics, Control and Measurements**

Department of Control and Instrumentation

Student: Bc. Martin Horák ***ID:*** 164286

Year of study: 2 ***Academic year:*** 2019/20

TITLE OF THESIS:

Semantic description of the embedded device screen

INSTRUCTION:

The aim of this work is to design and implement a system for recognition and analytical description of the embedded device display content based on machine learning. The input will be the cropped relevant region of the image provided by the camera that will contain only the embedded device screen. The output will be a semantic description of individual elements (e.g. input fields, buttons, text areas, etc.) on the screen.

1. Learn about machine learning in computer vision and define system output requirements.
2. Review existing machine learning methods and discuss the suitability and limitations of each solution. Select / suggest the most appropriate solution for given application.
3. Build a large and varied gallery of training and testing images.
4. Implement and test the selected procedure.
5. Process the results, determine the restrictive conditions, evaluate.

RECOMMENDED LITERATURE:

1. Goodfellow I., Bengio Y., Courville A.: Deep Learning. MIT Press, 2016. ISBN 9780262035613. (deeplearningbook.org)

2. Buduma, N., Locascio, N. Fundamentals of Deep Learning. O'Reilly Media, Inc. 2017. ISBN 9781491925614.

Date of project specification: 3.2.2020 ***Deadline for submission:*** 1.6.2020

Supervisor: Ing. Ilona Janáková, Ph.D.

doc. Ing. Václav Jirsík, CSc.
Subject Council chairman

ABSTRACT

This thesis deals with the detection of UI Elements in the image of a printer screen using the convolutional neural networks. The theoretical part of the thesis is a literature review of current object detection architectures. The practical part covers the creation of the dataset, training and evaluation of the selected models using the Tensorflow Object Detection API. The conclusion of the work discusses the suitability of the trained models for a given task.

KEYWORDS

Object detection, Convolutional neural networks, Tensorflow Object Detection API, Computer Vision

ABSTRAKT

Tato diplomová práce se zabývá detekcí prvků uživatelského rozhraní na obrázku displeje tiskárny za použití konvolučních neuronových sítí. V teoretické části je provedena rešerše současně používaných architektur pro detekci objektů. V praktické čísti je probrána tvorba galerie, učení a vyhodnocování vybraných modelů za použití Tensorflow ObjectDetection API. Závěr práce pojednává o vhodnosti vycvičených modelů pro zadaný úkol.

KLÍČOVÁ SLOVA

Detekce objektů, Konvoluční neuronové sítě, Tensorflow Object Detection API, Počítačové vidění

HORÁK, Martin. *Sémantický popis obrazovky embedded zařízení*. Brno, 2020, 102 p. Master's Thesis. Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Advised by Ing. Ilona Janáková, Ph.D.

Rozšířený abstrakt

Tato diplomová práce se zabývá detekcí prvků uživatelského rozhraní (např. tlačítko, text, obrázek, atd.) na obrázku displeje tiskárny za použití konvolučních neuronových sítí (KNS). Tato práce je součástí systému pro automatické testování tiskáren metodou černé skříňky za pomocí robotického ramena a kamery. Použití technik strojového učení je pro tuto aplikaci vhodné, jelikož uživatelské rozhraní různých výrobců se mohou výrazně lišit, což může být problematické pro klasické metody počítačového vidění. Vhodně natrénovaný model je schopný se naučit dostatečně obecnou definici jednotlivých tříd a tudíž detekovat i objekty, které jsou výrazně vizuálně odlišné od objektů v trénovací galerii.

Před vlastním řešením zadaného problému byl v první kapitole vysvětlen princip neuronových sítí, techniky učení a rozdíl od KNS. Dále byly vysvětleny základní techniky používané v konvolučních neuronových sítích a výpočet metrik pro evaluaci naučených modelů. Druhá kapitola pokračuje v budování na základech, které byly položeny v první kapitole, literární rešerší architektur pro klasifikaci obrazu pomocí KNS. Probrány jsou techniky použité jak ve starších průkopnických architekturách, tak i aktuálně používané techniky. Druhá část kapitoly vysvětluje princip fungování tří nejpoužívanějších architektur pro detekci objektů.

Jelikož se všechny zmíněné modely pro detekci objektů učí metodou s učitelem, třetí kapitola se zabývá veřejně dostupnými, anotovanými galeriemi obrázků a jejich předzpracováním. V kapitole jsou nejdříve představeny notoricky známé, obecné galerie pro počítačové vidění a poté je analyzována galerie dostupných obrázků obrazovek tiskáren. Jelikož zadaný úkol nespecifikuje třídy do kterých je nutno jednotlivé prvky uživatelského rozhraní klasifikovat, bylo nutné identifikovat elementární prvky uživatelského rozhraní. Výsledkem je seznam sedmi prvků (tlačítko, text, vstupní pole, obrázek, přepínač, zaškrtávací pole a posouvací přepínač), které byly následně detekovány. Jelikož dostupná galerie je značně limitovaná svou velikostí, byly zde diskutovány možnosti rozšíření pomocí veřejně dostupných galerií různých uživatelských prostředí. Během rešerše byly nalezeny dvě galerie Androidích aplikací, které byly následně použity pro rozšíření. Konec kapitoly se věnuje předzpracováním obrázků před učením včetně možnosti potlačení moiré obrazce, který vzniká interferencí matice pixelů snímané obrazovky a matice senzorů na kameře. Bylo vyzkoušeno několik metod odstranění moiré obrazce, avšak metoda, která dosahuje nejlepších výsledků je značně pomalá, což limituje její nasazení v praxi. Navíc bylo v jedné z pozdějších kapitol zjištěno, že odstranění tohoto obrazce nemá značný vliv na výslednou přesnost modelu, ale pouze na rychlost učení.

Čtvrtá kapitola práce se věnuje výběru platformy strojového učení, trénování vybraných modelů a vyhodnocení výsledků. Pro učení byla vybrána platforma

"Tensorflow Object Detection API", což je vysokoúrovňová knihovna postavená na platformě Tensorflow. Výhodou této knihovny je rozsáhlá nabídka před-trénovaných modelů a jednoduchá modifikace parametrů učení pomocí konfiguračního souboru. Nevýhodou je, že se tato knihovna nachází v "reseach" módu, což vede k nedostatečné dokumentaci a chybám při použití. Samotné trénování bylo rozděleno do dvou logických částí. V první části bylo vybráno deset před-trénovaných modelů z dostupné galerie modelů [60], které byly částečně natrénované na zlomku galerie. Z dosažených výsledků a z výsledků modelů z dokumentace byly vybrány čtyři modely, které pak byly plně vytrénovány v druhé části. V druhé části bylo také provedeno několik experimentů za účelem nalezení pseudo-optimálního nastavení trénovacích parametrů.

Testována byla rychlost detekce výsledného modelu, jelikož jedním z požadavků na výsledné řešení, je rychlost detekce v nejhorším případě ryhlostí jednoho snímku za sekundu. Rychlost modelů byla testována na počítači bez GPU jelikož nasazený model nebude mít k dispozici grafickou kartu. Výsledek testu potvrdil, že modely s architekturou *Faster R-CNN* jsou výrazně pomalejší než modely s architekturou *SSD* a tudíž je jejich použitelnost ve finálním řešení nepravděpodobná. Naměřené rychlosti *Faster R-CNN* se pohybovaly v rozmezí mezi 1.2 sekundy do 3 sekund zatímco modely požívající architekturu *SSD* dosahovaly rychlosti v rozmezí od 74ms do 297ms. Dalším provedeným experimentem byl, již zmíněný vliv moiré obrazce na přesnost učení. Tento test byl proveden na obou architekturách a ukázal, že odstranění moiré obrazce nemá výrazný vliv na přesnost sítě, avšak odstraněním značně urychlíme učení sítě. V dalším testu byl zkoumán vliv rozšíření galerie obrázků tiskáren obrázky Androidích obrazovek. Výsledek testu ukázal, že výsledné sítě se naučili opět na podobnou přesnost. Lze předpokládat, že rozšířený dataset se naučil robustnější model, zejména pro třídy, které nejsou v galerii zastoupeny tak často.

V páté kapitole byly vyhodnoceny čtyři vybrané vytrénované modely. Nejlepšího výsledku dosáhly modely *SSD inception v2* a *Faster R-CNN Resnet 50*, které dosáhly $mAP^{IoU=50} = 0.90$. Při zhodnocení dalších metrik zjistíme, že *Faster R-CNN* architektura předčila *SSD* při detekci malých objektů na co má pravděpodobně vliv i to, že *Faster R-CNN* přijímá na vstupu větší obrázek. Nevýhodou *Faster R-CNN* je rychlost, která je průměrně 3 sekundy, což je desetkrát pomalejší než *SSD*. Dále byly porovnány dva nejrychlejší modely ze seznamu dostupných před-trénovaným modelů *SSD Mobilenet v1 PPN* a *SSD Lite Mobilenet v2*. Z naměřených dat vyplývá, že oba modely dosahují podobných rychlostí při velikosti vstupního obrázku *300x300 px* avšak *SSD Lite* dosahuje daleko vyšší přesnosti $mAP^{IoU=50} = 0.83$ zatímco druhý model dosahuje pouze $mAP^{IoU=50} = 0.59$.

Na základě naměřených dat je v závěru vybraný model *SSD inception v2*, jakožto nejvhodnější kandidát pro danou úlohu z pohledu rychlosti a přesnosti detekce.

DECLARATION

I declare that I have written the Master's Thesis titled "Sémantický popis obrazovky embedded zařízení" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno 11.05.2020 .....

author's signature

ACKNOWLEDGEMENT

I would like to express my very great appreciation to Ing. Ilona Janáková Ph.D. for professional guidance, consultation, patience, and comments on my work. Furthermore, I would like to thank the ROBOT team at Y Soft who allowed me to work on this project.

Brno 11.05.2020 .....

author's signature

Contents

List of Figures

List of Tables

Introduction

Approximately about half a billion years ago, the Earth was mostly water with few species of animals floating around in the ocean. The life was very slow until something triggered what is called "evolutionary big bang". Within a geologically brief interval of perhaps 20 million years, virtually every modern animal phylum made its fossil debut. Approximately at the same time, the first animals developed eyes and the onset of vision most likely started rapid evolutionary "arms races" between predators and prey. [1]

In the 1960s started a computer vision evolution by trying to mimic parts of the complexity of the human vision system and enabling computers to identify and process objects in images and videos in the same way that humans do. Fast forward to 2010s, the computer vision has ridden the wave of improved deep learning techniques and became one of the most powerful and compelling types of AI. Nowadays, computer vision is becoming part of our everyday life. It is in our smartphone cameras making captured photos look better, it translates signs in a foreign language just by pointing your phone's camera, it is helpful in healthcare since a large portion of medical data is image-based, it unlocks our phones using the facial recognition and in the future, it will help to drive our cars.

The topic of this thesis is a classification of UI (User Interface) elements (e.g. button, text, checkbox, etc..) in the image of a screen. This work is a part of a larger system whose goal is automatic, black-box testing of any device with a touch screen or even device with physical buttons. Nowadays, it is used mainly for software testing of Y Soft SafeQ which is a third party management SW for multi-functional printers. The biggest bottleneck of new SW updates of HW devices is testing. On average it takes almost 14 days to manually test the new SW on printer which is not suitable with an agile style of product development. On top of that, the errors found during the testing phase are fixed, but the testing process does not start from the beginning so there is no warranty that the fix did not break some already tested part of the SW. The automatic, black-box testing using a robotic system can reduce the time needed for testing to two days by automatic test setup, continuous operation and simultaneous testing on multiple devices. Moreover, the overnight tests can provide fast feedback to a developer whether his change did not break some other part of the system.

The goal of this work is to further automate the creation of tests in this system. In the current version, the user creating a new test has to draw boundary boxes around all elements that will be used in the test. The proposed solution should be able to find all UI elements and classify them. There are two main requirements for the proposed solution. The first one is accuracy and the second one is the detection

speed that should not take much longer than one second.

In the future, the solution of this thesis can be used for exploratory testing which is a system that is able to create new test cases by itself which would be a peak of test automation.

1 Convolutional Neural Network

A Convolutional Neural Network (ConvNet/CNN) is a subset of neural networks that is specialized to work with spatially depended data such as images or sound. The agenda of this field is to enable machines to perceive the world similarly to humans. That means to extract the knowledge from the data and use it for a multitude of tasks such as image & video recognition, image analysis & classification, media recreation, recommendation systems, natural language processing, etc. This knowledge gives machines solid foundation to do more educated decisions based on the data and perform more complex tasks.

This chapter will explain the general concept of the neural networks and then it will continue with detail explanation of CNNs.

1.1 Neural Network

As you can see in the figure 1.1, the perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it returns binary output either bipolar "-1/+1" (see equation 1.3) or unipolar "0/1" output (see equation 1.2). That tries to mimic the behavior of the neuron in the brain which either "fires" or remains "silent" based on the input. The decision is made by summing up the weights which are multiplied by input data (including the bias and its weight w_0) and passed to an activation function. What makes this a "machine learning" algorithm is that it is able to **automatically learn the weights** based on the labeled data. By learning the weights it creates a model that can draw a linear decision boundary that allows us to discriminate between the two linearly separable classes.

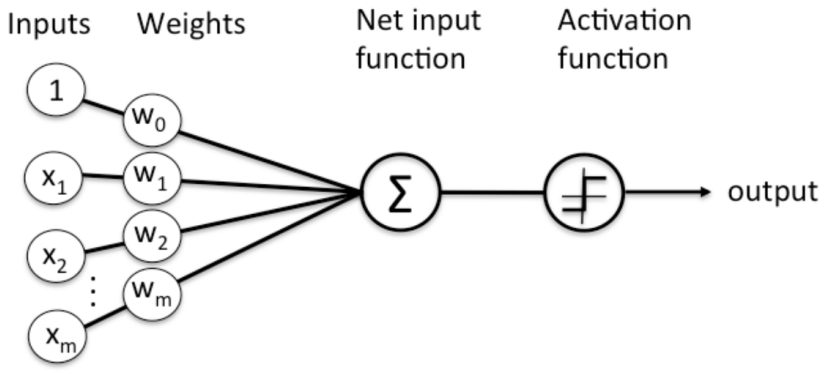


Fig. 1.1: Schematic of Rosenblatt's Perceptron [3]

In other words, the perceptron is a linear classifier defined by weights w_i, bias b and activation function $f(x)$ as can be seen in the equation 1.1.

$$y = f\left(w_0 + \sum_{i=1}^{n} w_i x_i\right) \tag{1.1}$$

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + w_0 > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + w_0 > 0 \\ -1 & \text{otherwise} \end{cases} \tag{1.3}$$

1.1.1 Neuron

Neurons that are used in neural networks work on the same principle as perceptrons with the only difference that the activation function is not bi-polar, but continuous e.g. *sigmoid*, *tanh*, *ReLU*, etc. The goal of the neural network is to group neurons into multiple layers in order to increase representational capabilities. Figure 1.2 shows neurons grouped into multiple layers. Each layer consists of a number of neurons and each neuron is connected to all neurons of the following layer. However, neurons of the same layer are not connected to each other. Each of this connection has associated weight w_i, j and each neuron has own bias b. These weights and biases are learned during the learning process, this results in a network that can be considered as a universal approximator, i.e. it can approximate any continuous function, and thus do not suffer from the same limitations as a single perceptron.

The size of the network (number of layers and neurons in each layer) is chosen based on the complexity of the task that we want to approximate. It can be difficult to design a good architecture, as using too many units and layers may result in a complex function that overfits training data, and choosing too few units can produce a biased function that is too simple to represent the data distribution well.

1.1.2 Activation Function

The nonlinear activation function is essential for improving the approximation quality of the neural network. Stacking neurons without a non-linearity would not achieve any improvement because a combination of several linear transformations can be always simplified into a single linear transformation. Furthermore, real-world problems require non-linear solutions which are not trivial. Therefore, it is necessary to use non-linearity which is done with an activation function. The figure 1.3 illustrates most common activation functions. *Sigmoid* and *tanh* used to be very

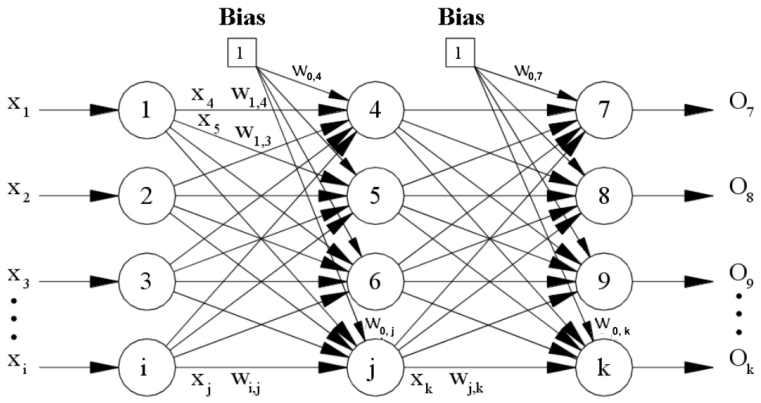


Fig. 1.2: Neural Network With One Hidden Layer [6]

popular in the past, but nowadays most of the networks use *ReLU* because it is computed much faster and it does not have a problem with *vanishing gradient*.

The vanishing gradient arises due to the nature of the back-propagation optimization where gradients tend to get smaller and smaller as we keep on moving backward. This is particularly problematic with *Sigmoid* and *tanh* because both functions are nearly flat for input value x that is either too high or too low. This causes vanishing gradients and poor learning for deep networks.

The downside of *ReLU* is a problem called "dying ReLU" which is caused by zero output of the function for all negative values. This problem is addressed by a *ReLU* variations *"Leaky ReLU"* and "*ELU*" which have small negative slope for negative values.[7]

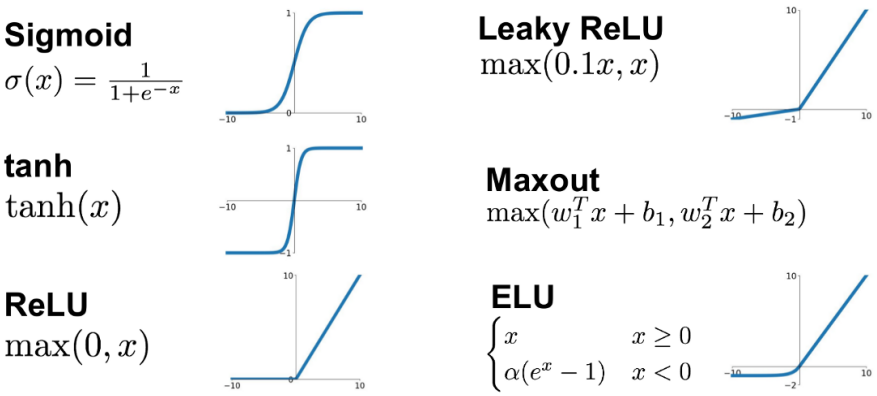


Fig. 1.3: Activation Functions [7]

1.1.3 Cost Function

Cost function (sometimes referred as *"loss function"*) measures performance of the learned model for given data. It quantifies the error between predicted values and expected values and presents them in the form of a single real number. Depending on the type of the function we are searching for a candidate solution that has the highest or lowest score. The return value of cost function that should be minimized is usually called *"cost"*, *"loss"* or *"error"*. The goal is to find the values of model parameters for which the cost is as small as possible. In the second case, the returned value is usually called "reward" and the aim is to maximize this value. There's no universal cost function that suits all the problems. There are a lot of factors involved such as the type of machine learning algorithm, difficulty of calculating the derivatives and to some degree the percentage of outliers in the data set.

In case of regression, we can use e.g. **MSE**(Mean Square Error), which measures the average squared difference between predictions y_i and actual observations $\hat{y}_i$. It is easy to calculate, but it heavily penalizes prediction that are far away from actual values due to the squaring.[8]

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} \tag{1.4}$$

For classification is a very popular **Cross Entropy Loss**. The cost of this function increases as the predicted probability diverges from the actual label. An important aspect of this cost function is that it penalizes heavily the predictions that are confident but wrong.

$$\text{CrossEntropyLoss } = -\left(y_i \log\left(\hat{y}_i\right) + (1 - y_i) \log\left(1 - \hat{y}_i\right)\right) \tag{1.5}$$

1.1.4 Backpropagation

Backpropagation, is an algorithm for supervised learning of neural networks using gradient descent. It is an iterative process that aims to lower the cost function by fine-tuning the weights of the model.

Even though this method was used since the late 1950's it was mostly limited to linear systems. One of the significant reasons that helped this algorithm to become the backbone of machine learning is the realization that local minima in loss function might not be a significant problem which was proved by the success of gradient-based learning techniques such as Boltzmann machines and others that used BP to compute the gradient for non-linear systems. The fact that the local minima are not a significant problem is due to oversized neural networks that are

commonly used in practice. The extra dimensions in these networks reduce the risk of unreachable regions.[9]

The figure 1.4 explains on simplified example with only one weight w how gradient descent works. As stated before, the goal of the algorithm is to find the minimum of the cost function $J(w)$. The principle of the algorithm can be described as "climbing down a hill" until a local or global minimum is reached. At each step, we calculate the gradient and make a step in the opposite direction in order to "go downhill". The size of the step is determined by the *learning rate* as well as the slope of the gradient. The learning rate is another hyper-parameter that can be tuned. The low learning rate causes slow convergence to the minimum and it can potentially cause the algorithm to get stuck in the local minima. If we choose the learning rate too big it can never find the minimum because it will always overshoot.

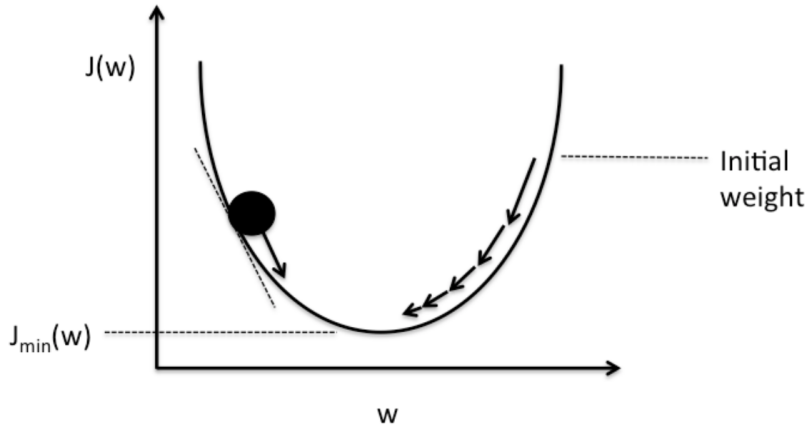


Fig. 1.4: Schematic of Gradient Descent [3]

Mathematically it is described as partial derivative of the cost function J for each weight w_j in the weight vector multiplied by learning rate α (see equation 1.6)

$$\Delta w_j = -\alpha \frac{\partial J}{\partial w_j} \tag{1.6}$$

1.2 Convolutional Neural Network

Convolutional Neural Network differs from the classical neural network in the size of the receptive field. In multi-layer NN, every neuron is connected to all neurons in the following layer while in CNN the neuron is connected with just a limited neighborhood of neurons in the following layer. This design decision results in a series of advantages and disadvantages/restrictions. The most crucial restriction is that this kind of network is suitable only for the data with a spatial or temporal

dependency e.g. images or sound. In case of images, the spacial dependency means that pixels that are close together carry in general more information than pixels that are further away. The idea of connecting units to local receptive fields on the inputs goes back to the perceptron in the early 60s and was almost simultaneous with Hubel and Wiesel's discovery of local sensitive, orientation sensitive neurons in cat's visual system.[10]

1.2.1 Convolution Layer

Convolution is mathematically defined as a "mathematical operation on two functions (f and g) that produces a third function expressing how the shape of one is modified by the other". It is defined as the integral of the product of the two functions after one is reversed and shifted (see equation 1.7)[11].

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{1.7}$$

In case of image processing, we use a discrete 2D convolution that can be imagined as a kernel that "slides" over the image (2D input data), performing an element-wise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel (see the figure 1.5). Mathematically it is described by the equation 1.8 where $g(x, y)$ is filtered image, $I(x, y)$ is the input image and K is the filter kernel.

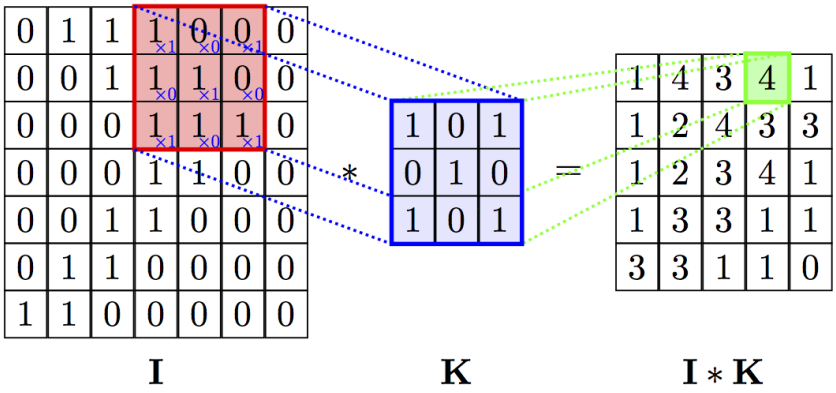
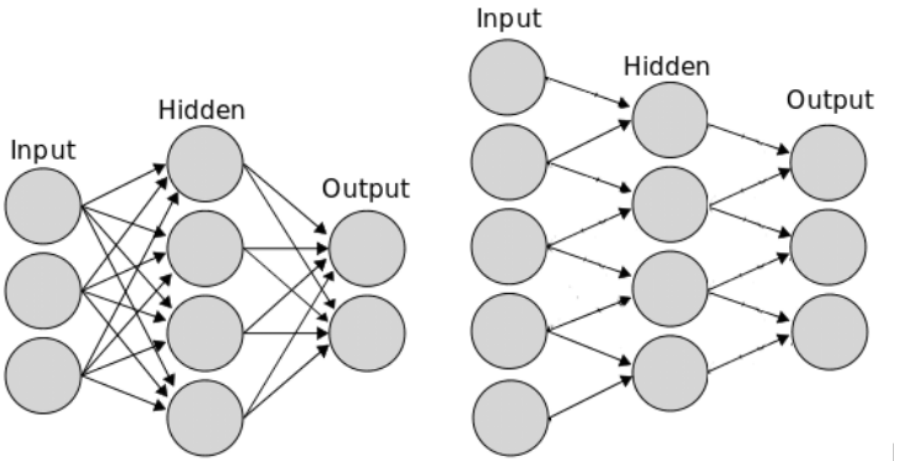


Fig. 1.5: Visualization of 2D convolution [4]

By using a single kernel that "slides" over the whole image we also drastically reduce the number of parameters in the network because the weights are shared over the whole image. The figure 1.6 describes this principle in simplified one dimensional scheme of the CNN. The saved capacity can be used more efficiently by using more filters (which results in more channels of feature map) and extract more features from the image.



(a) Fully Connected NN (b) CNN

Fig. 1.6: Connections in fully connected NN and CNN [5]

Another advantage of using the convolution is a shift invariance that is also caused by the "sliding" kernel over the image. The weights of the filter are shared across all patches of the image so the weights learned will be invariant to position.

$$g(x,y) = I * K(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} I(s,t)K(x-s,y-t) \tag{1.8}$$

By using the convolutional layers, the network is able to learn to recognize local features, such as edges and corners. For example, the filters G_x and G_y in the figure 1.7 are used to detect horizontal and vertical edges. After applying both of the filters to the image we get the image with emphasized edges which can bee seen in the figure 1.8. The CNN usually consists of many hidden layers. The layers close to the beginning of the network usually learn kernel weights to extract elementary visual features for example G_x and G_y kernels for edges. Later in the deeper layers of the network are these elementary visual features combined into more complex features such as geometrical shapes.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Fig. 1.7: Sobel Convolution Kernels

Fig. 1.8: Result of Applying Sobel Filter

The figure 1.9 shows the typical architecture of CNN for image classification. It consists of alternating convolutional layers and pooling layers. Convolution layers take the inner product of the linear filter and the underlying receptive field followed by a nonlinear activation function at every local portion of the input. The resulting outputs are called feature maps whose **number of channels is equal to the number of used filters**. The CNN below has on input 224x224 RGB image on which 64 filters are applied. This results in a feature map with the same dimensions (due to the padding) as the input image but with 64 channels. In the next step, max-pooling is used (explained in the following section) to decrease the dimensions from 224x224 to 112, followed by a convolution with 128 filters. The reduction of dimensions and increase of channels is used multiple times to increase expressiveness of the network. The network ends with three fully connected layers that classify extracted features into 1000 classes.

What makes the architecture of CNN networks so exceptional in comparison with traditional pattern recognition methods, is the backpropagation that is able to figure out what features in the image are helpful and learn the weights of the convolutional filters to recognize them alone from the data. In the traditional pattern recognition, a hand-designed feature extractor gathers relevant information from the input. The trainable classifier then categorizes the features vector into classes. The classifier can be implemented as for example fully connected NN, KNN, Naive Bayes, etc. Extracting more complex features using a hand-designed extractor is tedious, especially in case of image processing. To extract features from the image it is also possible to use the fully-connected neural network, but the resulting network would be too large because image data can contain easily hundreds of pixels that would be an input to the network.[9]

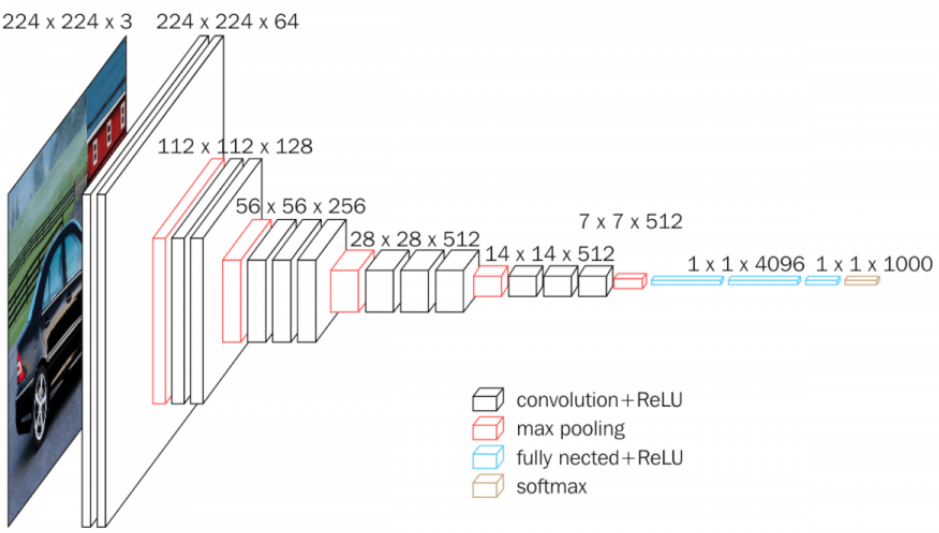


Fig. 1.9: VGG-16 Architecture

1.2.2 Pooling Layer

The pooling layer, a common layer in CNNs, is used to reduce the spatial size of the feature map which results in a reduction of parameters, computations in the network, and hence to controls overfitting. This layer does not have any learnable weights, it is typically a patch of size 2x2 that is applied to each channel separately with a stride 2 along both width and height of feature map, discarding 75% of activations (see the figure 1.10). The most used type of pooling is Max Pooling, which simply selects the maximum value in a channel within the patch (see image below). Another option is Average Pooling which averages values within the path.

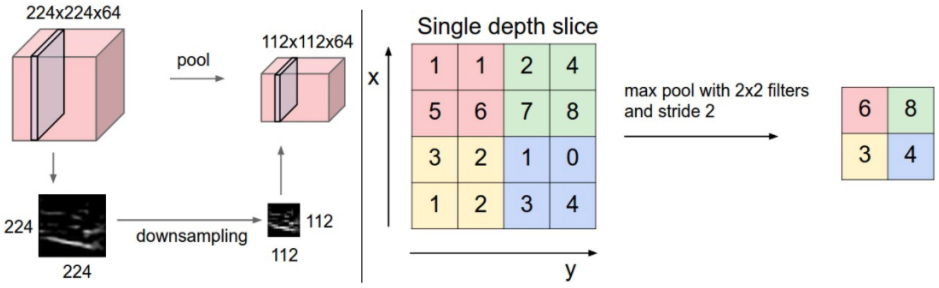


Fig. 1.10: The principle of max pooling [12]

There was a discussion whether it is better to use pooling or convolution with stride higher than 1 to reduce the size of the feature map. The assumption was that convolution has learnable weights so it is not thoughtlessly discarding the data. This assumption was proved false as pooling seems to achieve slightly lower error in most of the cases and does not increase the number of parameters in the network.[13]

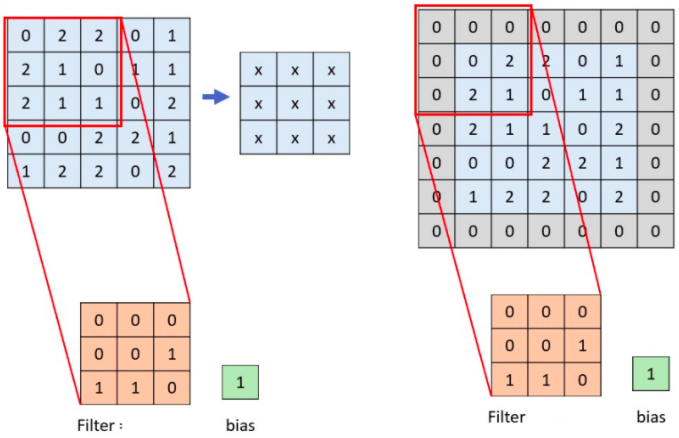


Fig. 1.11: Zero Padding [4]

1.2.3 Zero Padding

This technique is employed to prevent the convolution from reducing the spatial size of the output feature map and losing information on corners of the image. It is done simply by adding an additional layer of zeros around the image (see Fig. 1.11).

1.3 Overfitting

Thanks to the increase of computational power, it is possible to use neural networks with sometimes tens of millions of parameters. That allowed us to make tremendous progress in areas such as image recognition, object detection or natural language processing. With such a capacity of the network comes also a weakness. The network becomes capable to memorize the dataset instead of deriving a general model. This can be spotted by comparing the training and testing accuracy (see Fig. 1.12). When the accuracy of the network on the training dataset is much better or starts to diverge from the validation/test dataset, it is a sign of overfitting. The commonly used methodologies to avoid overfitting are for example cross-validation, early-stopping, pruning and regularization. [14]

Underfitting is on the other hand a case when the trained model is too simple to capture the complexity of the data. For example, using a linear model for image recognition will generally result in an underfitting model. To avoid underfitting, it is possible to increase the capacity of the network by adding more hidden layers, changing regularization parameters or learning rate.

Both underfitting and overfitting are very well visualized in the figure 1.13.

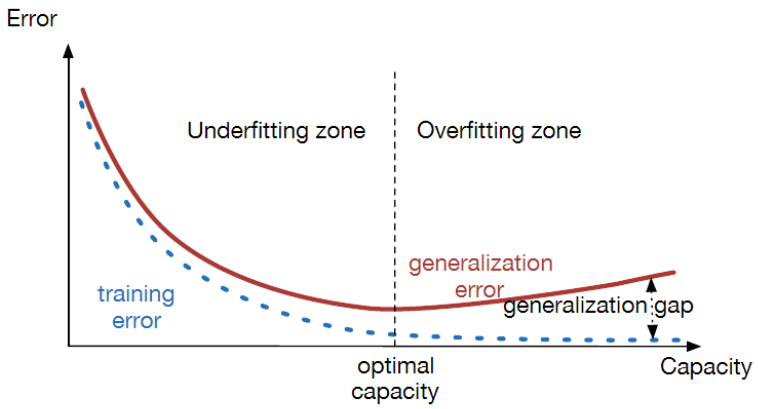


Fig. 1.12: Model Capacity and its Effect on Underfitting and Overfitting [14]

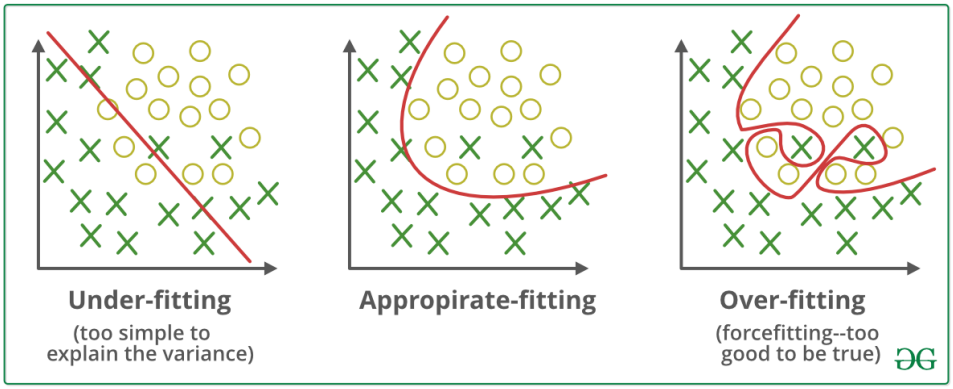


Fig. 1.13: Visualization of Underfitting, Overfitting and Appropriate Fitting [15]

Regularization is a technique used to prevent overfitting and underfitting problems. This chapter describes techniques that are commonly used and were proven to be effective. The whole subchapter is heavily inspired by [14].

1.3.1 Early Stopping

Early stopping is the simplest and most effective way of preventing overfitting. The basic idea is to measure the validation loss and stop the training when it starts to increase instead of decrease. The principle is illustrated in the figure 1.12.

1.3.2 L2 Regularization

L2 Regularization (sometimes referred to as "Weight Decay") is a commonly used technique in machine learning. It works by adding a Regularization Term to the cost function. The regularization term consists of the sum of squares of all the

link weights in the network, multiplied by a parameter λ called the Regularization Parameter. Note that the Regularization Term does not include the biases since in practice it has been found that their inclusion does not make much of a difference to the final result.

$$\text{Loss} = \text{Error}(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2 \tag{1.9}$$

This regulation leads to more diffused weight parameters which encourage the network to use all its inputs a little rather than some of its inputs a lot. That helps because overfitted models tend to rely on the model parameters that blow up in the value during training. That causes the model to give too much importance to the idiosyncrasies of the training dataset.

1.3.3 L1 Regularization

L1 Regulation is very similar to the L2 Regulation. The main difference is that it uses the sum of the absolute value of the weights instead of the quadratic function used in L2 Regulation, which also leads to smaller more evenly distributed weight values.

$$\text{Loss} = \text{Error}(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i| \tag{1.10}$$

In the L2 Regulation the weight reduction is proportional to the weight value and "punishes" higher weight values more than lower values. L1 Regulation on the other hand reduces weights by fixed amount in every iteration. As a result L1 Regularization leads to networks in which the weight of most of the connections tends towards zero, with a few connections with larger weights.

1.3.4 Dropout Regularization

This is one of the most common and most effective techniques that are used. The basic idea is to randomly drop a certain percentage of neurons each iteration of the Backpropagation algorithm. That means, that during the training the network uses only a subset of the neurons and during the testing, the network uses all neurons. To keep the average activation values the same as during the training phase, the activations are usually multiplied by a probability of the node being shitted down during the training.

During Dropout training, a hidden node cannot rely on the presence of other hidden nodes in order to be able to accomplish its task. Hence each hidden node must perform well irrespective of the presence of other hidden nodes, which makes

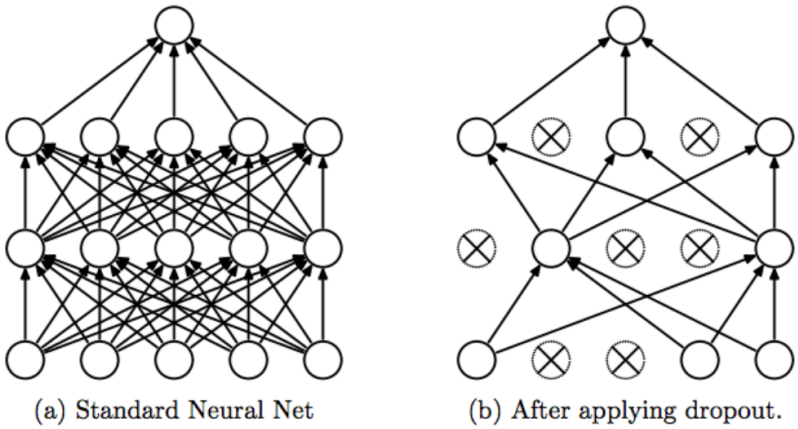


Fig. 1.14: Dropout Regularization [14]

them very robust and work in a variety of different networks, without getting too adapted to any one of them. This property helps in generalizing the classification accuracy from training to test data sets.

1.3.5 Training Data Augmentation

The bigger the network we want to use the bigger dataset is needed to ensure proper training and avoid overfitting. The problem with small datasets can be addressed by artificially expanding the training set by applying for example geometrical transformations (e.g. translation, rotation), color transformations (e.g. changing contrast or brightness) or by adding some random noise.

1.3.6 Batch Normalization

To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. To calculate the standard deviation it adds two trainable parameters to each layer, so the normalized output is multiplied by a standard deviation parameter γ and add a "mean" parameter β

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Fig. 1.15: Batch Normalizing Transform Applied to Activation x Over a Mini-batch [16]

Batch normalization enables to use higher learning rates (in the non-normalized network, it would lead to oscillations), better gradient propagation through the network (so it is possible to use more hidden layers) and it helps to reduce strong dependencies on the parameter initialization values.

1.3.7 K-Folds Cross Validation

Cross validation is a model evaluation method that can be used to train a less biased model with limited input data by ensuring that every observation from the original dataset appears at least once in training and test dataset. This method follows the following steps:

1. Split the entire dataset randomly into k folds. The value of k should not be too high otherwise the test set would be too small.
2. Train the model using the $k-1$ folds and validate the model using the remaining fold.
3. Repeat this process until every k-fold serves as the test set. Then calculate the model metric by averaging the model scores on each tested k-fold.

1.4 Object Detection

Image classification predicts what objects are present in the image but it does not provide any information about the positions of the objects in the image. This problem is tackled by the object detection algorithms that instead of predicting object class from an image, it predicts the class as well as a rectangle (called bounding box) containing that object. To locate the objects in the image, we have to predict four variables that uniquely define rectangle for each object in the image.

The most straight forward approach is to crop a fixed size "window" from the input image and feed all these patches to the image classifier. The biggest problem of this approach besides the computational complexity is the fixed window size that limits us in terms of size of the objects in the image. This is solved by resizing the image at multiple scales and run the object detector with the fixed window size on each of these resized images. The resizing is usually implemented as down-sampling of the original image by factor 2 until the minimum image size is reached.

Another problem is the varying aspect ratio of objects for example a sitting and a standing person. This problem is addressed by more advanced algorithms that will be discussed in the chapter 2.2.

1.5 Evaluation

The performance of the object detection model is evaluated by metrics that allow us to objectively compare multiple detection systems.

This chapter will explain the most common metric of choice used in the popular competitions such as PASCAL VOC, ImageNet, and COCO challenges — *The mean Average Precision (mAP).*

1.5.1 Intersection Over Union

Intersection over Union (IoU) is an evaluation metric used to measure the accuracy of a detected bounding box and the ground-truth bounding box. It is defined as the area of the intersection divided by the area of the union of a predicted bounding box B_p and ground-truth box B_gt. See the visual representation of the equation in the figure 1.16.

$$IoU = \frac{area\,(B_p \cap B_{gt})}{area\,(B_p \cup B_{gt})} \tag{1.11}$$

1.5.2 Predictions

Before defining various types of predictions that can be made by a classifier we have to define another metric called Confidence Score. The Confidence Score says how certain is the classifier that the proposed boundary box contains some object (regardless of class). In other words, it is a probability that the boundary box contains an object.

Based on the confidence score, IoU and predicted class we differentiate between the following types of predictions:

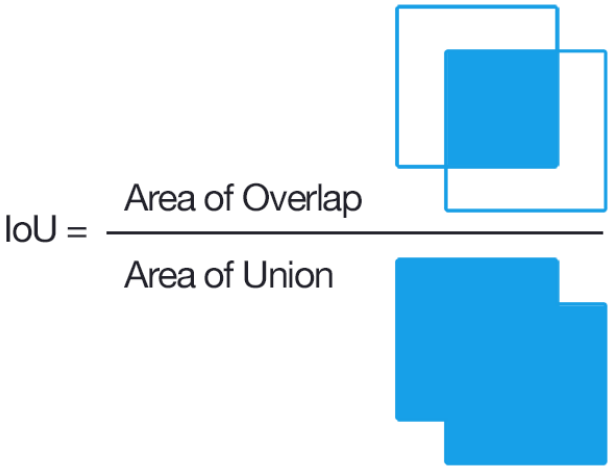


Fig. 1.16: Computing the Intersection Over Union [17]

1. **True Positive (TP)** prediction satisfies three conditions: $confidencescore > threshold$, the predicted class matches the class of a ground truth, the predicted bounding box has an $IoU > threshold$;
2. **False Positive (FP)** prediction has $confidencescore > threshold$ but the predicted class is wrong or $IoU > threshold$;
3. **False Negative (FN)** is a prediction with a confidence score of detection that is supposed to detect a ground-truth is **lower** than the threshold - detector does not detect the object in the location where the object is present;
4. **True Negative (TN)** is prediction where confidence score of a detection that is not supposed to detect anything is lower than the threshold - detector correctly does not detect the object in the location where is no object.

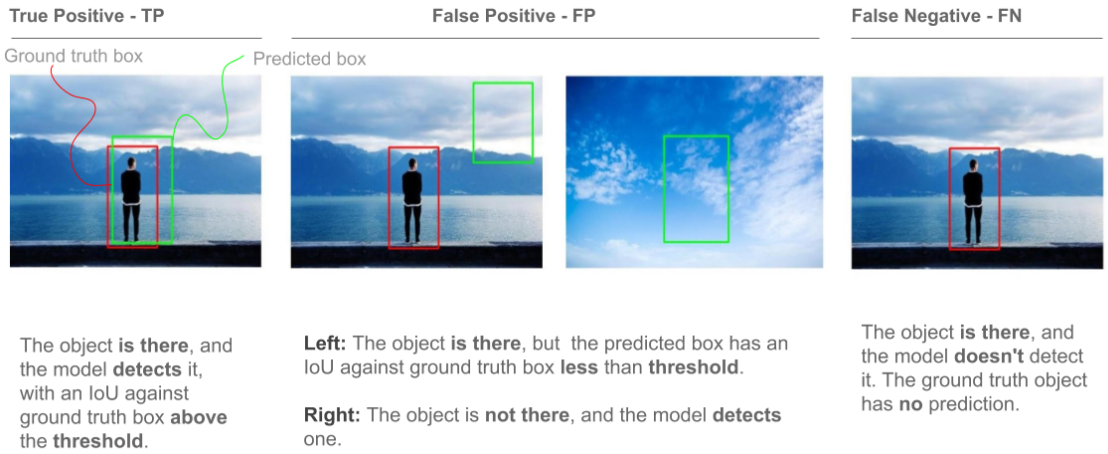


Fig. 1.17: True Positive, False Positive and False Negative Predictions [18]

By classifying predictions into those four categories we can calculate *accuracy*, *precision* and *recall*.

Accuracy is the percentage of correctly predicted examples out of all predictions. This metric is not used often because it can be very misleading when dealing with class imbalanced data as it puts more weight on learning the majority classes than the minority classes.

$$\text{accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{1.12}$$

Precision is defined as the number of true positives divided by the sum of true positives and false positives

$$\text{precision} = \frac{TP}{TP + FP} \tag{1.13}$$

Recall is defined as the number of true positives divided by the sum of true positives and false negatives where the denominator is the sum of ground-truths so we do not have to count the number of FN.

$$\text{recall} = \frac{TP}{TP + FN} \tag{1.14}$$

1.5.3 Precision x Recall Curve

The precision x recall curve is a suitable measure to assess the model's performance on imbalanced datasets. It plots recall on the x-axis and precision on the y-axis, where each point in the curve represents recall and precision values **for a certain confidence value**. Figure 1.18 shows that with increasing recall the general tendency of the precision is to decrease.

With decreasing threshold for confidence score, recall increases and precision decreases.

1.5.4 Average Precision

Because it is much easier to compare numerical values rather than plots when evaluating multiple models, we usually use the **average precision (AP)** which is a numerical value based on the precision-recall curve. It is simply calculated by averaging precision across all unique recall levels. To reduce the impact of wiggles in the curve it is firstly interpolated using the equation:

$$p_{\text{interp}}(r) = \max_{r' \geq r} p\left(r'\right) \tag{1.15}$$

Average precision is then calculated as the area under the interpolated precision-recall curve using the equation:

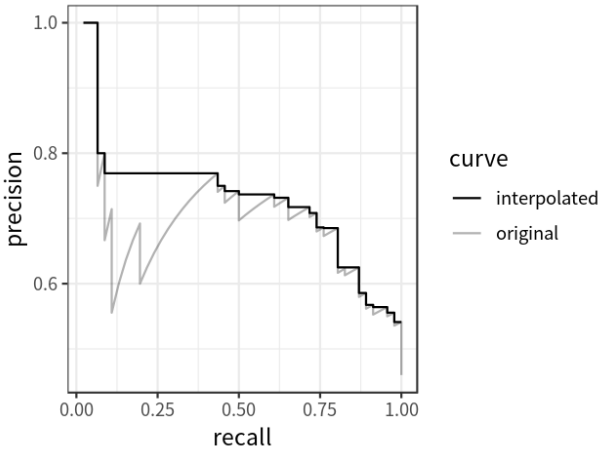


Fig. 1.18: Precision Recall Curve [36]

$$AP = \sum_{i=1}^{n-1} (r_{i+1} - r_i)\, p_{interp}\,(r_{i+1}) \tag{1.16}$$

where n is number of recall levels.

1.5.5 Mean average precision

Mean average precision (mAP) is average of APs over K classes:

$$mAP = \frac{\sum_{i=1}^{K} AP_i}{K} \tag{1.17}$$

1.5.6 Average recall

Average recall (AR) is calculated in simmilar manner as AP, but instead of calculating at a particular IoU, we calculate the AR at IoU thresholds from 0.5 to 1 and thus summarize the distribution of recall across a range of IoU thresholds.

$$AR = 2 \int_{0.5}^{1} recall(IoU) \mathrm{d}IoU \tag{1.18}$$

1.5.7 Mean average recall

Mean average recall (mAR) is average of ARs over K classes:

$$mAR = \frac{\sum_{i=1}^{K} AR_i}{K} \tag{1.19}$$

1.5.8 The COCO evaluation metrics

I have decided to use COCO evaluation metrics which is a combination of AP and AR with various thresholds of IoU, object size and number of detections which gives detailed description of model capabilities. The figure 1.19 describes all 12 metrics are used for characterizing the performance of an object detector.

Average Precision (AP):
- AP % AP at IoU=.50:.05:.95 **(primary challenge metric)**
- $AP^{IoU=.50}$ % AP at IoU=.50 (PASCAL VOC metric)
- $AP^{IoU=.75}$ % AP at IoU=.75 (strict metric)

AP Across Scales:
- AP^{small} % AP for small objects: area < 32^2
- AP^{medium} % AP for medium objects: 32^2 < area < 96^2
- AP^{large} % AP for large objects: area > 96^2

Average Recall (AR):
- $AR^{max=1}$ % AR given 1 detection per image
- $AR^{max=10}$ % AR given 10 detections per image
- $AR^{max=100}$ % AR given 100 detections per image

AR Across Scales:
- AR^{small} % AR for small objects: area < 32^2
- AR^{medium} % AR for medium objects: 32^2 < area < 96^2
- AR^{large} % AR for large objects: area > 96^2

Fig. 1.19: Metrics used in COCO Challenge [35]

1.5.9 Top-N score

Top-N score is another commonly used evaluation metric. The main difference of this method is that it takes into account N best predictions instead of calculating only with the best prediction.

For example the Top-1 accuracy is a conventional way to calculate the accuracy using only the prediction with the highest score. The Top-5 accuracy considers a prediction correct when the ground-truth class is any of 5 highest probability predictions. This is useful especially for tasks with a large number of classes.

1.5.10 Confusion Matrix

A confusion matrix is a technique for summarizing the performance of a classification algorithm. It can give us a better idea of what is the classification model getting right and what types of errors it is making. As you can see in figure 1.20, the rows represent the predicted classes and the columns represent the ground-truth classes. This results in a matrix where the numbers on the diagonal represent the correct predictions (e.g. cat was correctly classified four times). The rest of the numbers in the matrix represent wrong classifications. For example, the number 6 in the first row and second column (intersection of cat and fish class) says that six images with

fish were falsely classified as a cat by the model. This can help to find classes that look similar to the model.

		True/Actual		
		Cat	Fish	Hen
Predicted	Cat	4	6	3
	Fish	1	2	0
	Hen	1	2	6

Fig. 1.20: Example of Confusion Matrix [45]

In some cases the matrices contain one more column and row representing a "null" class. This class contains a number of predictions that did not find any of the classes with sufficient confidence in the image. The confusion matrix can also use relative numbers instead of absolute prediction count used in the example above.

2 Architectures

The architecture of convolutional neural networks has undergone great improvements over its lifetime. Especially in recent years, we have witnessed the birth of numerous CNNs that push the accuracy in the computer vision challenges. This chapter will briefly introduce some historical architectures as well as the state of the art architectures that are currently used.

2.1 Classification

Image classification is likely the most well-known problem in computer vision and it is the foundation for more complex tasks such as object detection. This sub-chapter will present some of the architectures from both past and present.

This chapter uses a simplified schemes of network architectures form [19]. The figure 2.1 shows legend for architectures used in this chapter.

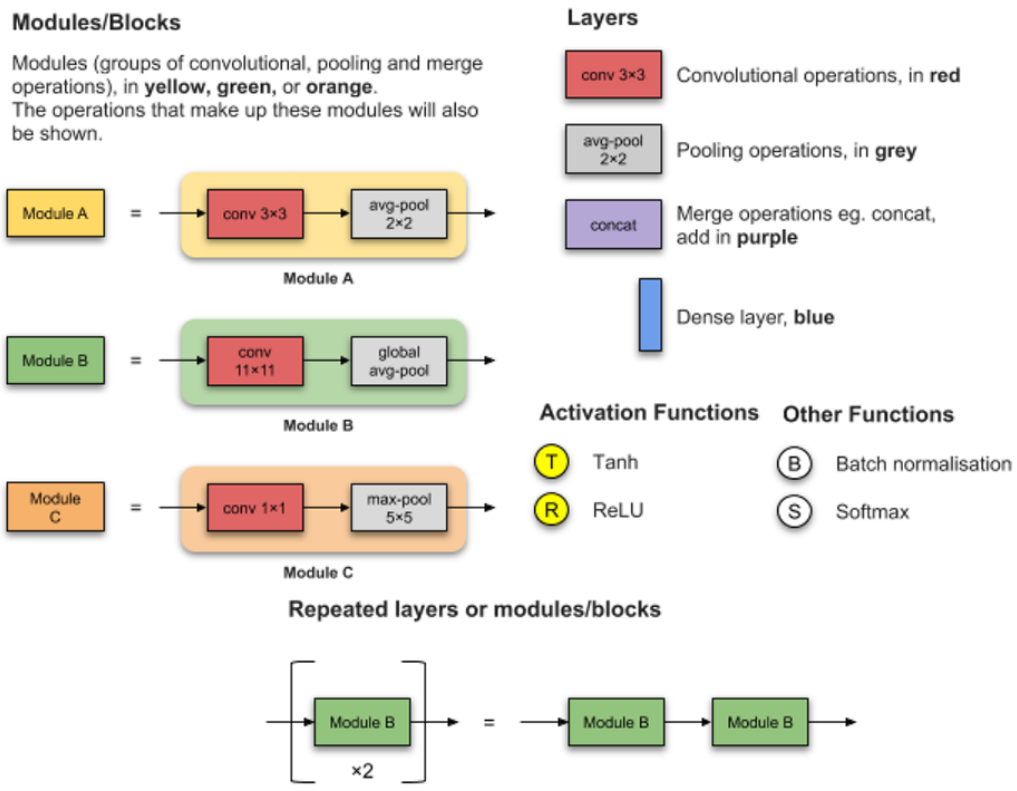


Fig. 2.1: Legend for Architecture Schemes [19]

2.1.1 Neocognitron

This architecture is most likely the great-grandparent of convolutional networks that are used nowadays. It was proposed in 1979 by Kunihiko Fukushima. His paper is surprisingly based on research done by neurophysiologists (Hubel & Wiesel, 1962) who found out that in the visual area of the cerebrum, neurons respond selectively to basic local features such as lines and edges. Then in the area higher than the visual cortex were found cells that respond to simple shapes such as circle, triangle, square or even to the human face. On top of that, these feature extractors are not complete at birth, but they are gradually developed after birth.[37]

Inspired by the visual cortex of the mammals, Fukushima designed a network that has 8 layers. The architecture uses two types of layers. Layers of "S-cells" that work as feature extractors and layers of "C-cells" which allow for positional errors in the features of the stimulus. These two types of layers are alternating throughout the network. Each C-cell receives output from a group of S-cells that extract the same feature from a slightly different position and the C-cell gets activated in the case at least one of the incoming S-cell is activated. [37]

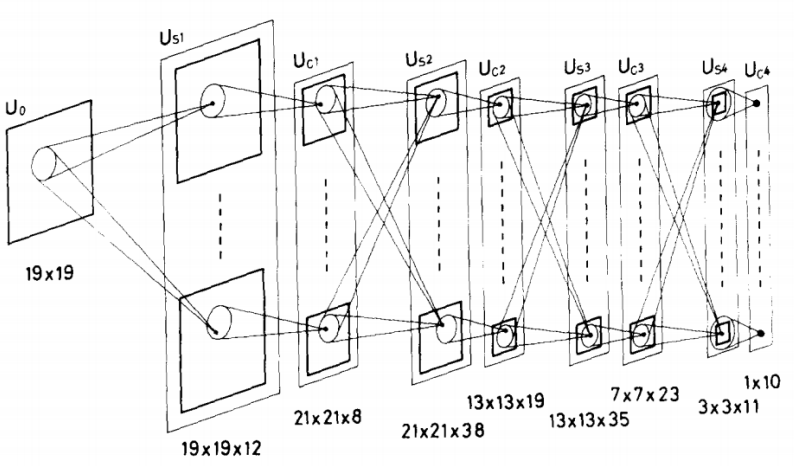


Fig. 2.2: Hierarchical Network Structure of the Neocognitron. (The numerals at the bottom of the figure show the total numbers of S- and C-cells in individual layers of the network which are used for the handwritten numeral recognition system) [37]

This network architecture has been used for handwritten character recognition and other pattern recognition tasks. I have decided to mention it because it implements all the fundamental ideas of CNN (local receptive fields, shared weights, and spatial sub-sampling) except the backpropagation instead of which Fukushima used his own function for supervised learning.

2.1.2 LeNet

This is a pioneering 5-layer Convolutional network by Yann LeCun published in 1998, that was developed for classification of handwritten digits on bank checks in the United States. It consists of 2 convolutional layers followed by 3 fully connected layers. The input to this network is 32x32 binary image which results in about 60k parameters in the whole network. It is stacking pooling layers in between the convolutional layers in order to reduce the spatial size of the feature map, reduce the number of parameters and computations in the network. This approach became a standard template for more complicated architectures.[9]

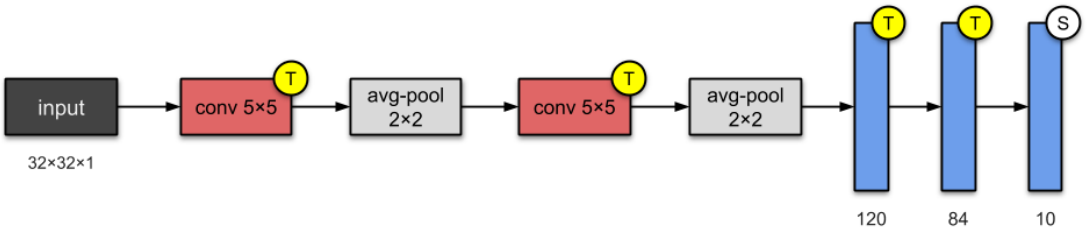


Fig. 2.3: LeNet-5 Architecture [19]

At the time of publication of LeNet, image recognition tasks have been done mostly by using hand-designed feature extractors followed by a classifier. LeNet came with an innovative approach where the network learns feature extractors alone from the input data which allows creating more complex extractors automatically and made the creation of extractors by hand redundant.

2.1.3 AlexNet

Alexnet is a way bigger network than LeNet, published in 2012 by Alex Krizhevsky. The ambition of this architecture was to design a network with a capacity big enough to classify images into 1000 classes in ImageNet LSVRC-2010 contest.

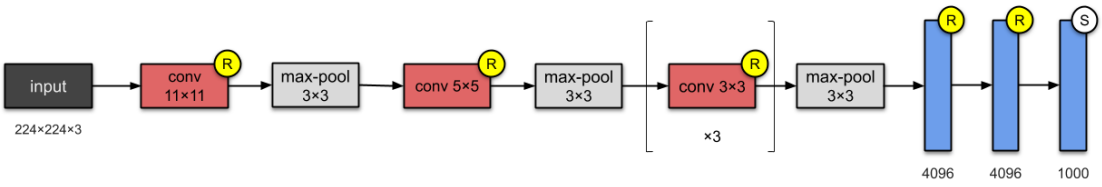


Fig. 2.4: AlexNet Architecture [19]

The proposed neural network consists of 8 layers - 5 convolutional and 3 fully connected. The input is 224x224 color image which results in a network with 60 million parameters (which is a thousand times more than LeNet). Besides the mind-blowing increase in the size of the network, this network uses several new techniques.

First of all, it uses *Rectified Linear Units (ReLU)* instead of a *tanh* function that was commonly used up to this point. The advantage of ReLU is that it prevents the vanishing gradient problem so the network is trained faster (approx. 6 times faster than *tanh* activation on CIFAR-10 dataset).

Secondly, AlexNet was designed so it allows for multi-GPU training by putting half of the model's neurons on one GPU and the other half on another GPU. This reduces training time and allows for a larger model.

Another innovation was in pooling. Traditionally, the pooling is done without overlapping of pooling kernels. This architecture uses a pooling kernel of size 3x3 with step 2 which results in overlapping which reduced the top-1 and top-5 error rates by 0.4% and 0.3% respectively.

Despite the enormous size of the dataset, there was a major problem with over-fitting that was tackled by using data augmentation and dropout. [20]

2.1.4 VGG Net

VGG is a network published in 2014 by Karen Simonyan, Andrew Zisserman named after the department of Oxford University - Visual Geometry Group, where they worked. As mentioned in the abstract of their paper, the main goal was to investigate the effect of the network depth on its accuracy. They evaluated 6 networks with depth ranging from 11 to 19 layers (including 3 fully connected layers at the end of the network). The largest network, with the 19 layers has 144 million parameters.

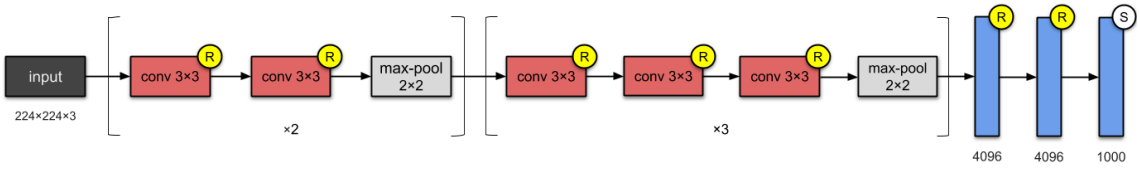


Fig. 2.5: VGG-16 Architecture [19]

The VGG network comes with novel design decision to limit the maximal size of convolution kernel to 3x3 (AlexNet uses 11x11 and 5x5 kernels in the first two layers). This architecture also utilizes 1x1 convolution filters, which can be seen as a linear transformation of the input channels (followed by nonlinearity). Smaller filters allow reducing the number of parameters and increase the number of layers and filter channels.

ConvNet config.	smallest image side		top-1 val.	top-5 val.
	train (S)	test (Q)	error (%)	error (%)
A (11 layers, 133M params)	256	256	29.6	10.4
A-LRN (11 layers, 133M params)	256	256	29.7	10.5
B (13 layers, 133M params)	256	256	28.7	9.9
C (16 layers, 134M params)	256	256	28.1	9.4
D (16 layers, 138M params)	384	384	28.1	9.3
	[256; 512]	384	27.3	8.8
E(19layers, 144Mparams)	256	256	27.0	8.8
	384	384	26.8	8.7
	[256; 512]	384	25.6	8.1
	256	256	27.3	9.0
	384	384	26.9	8.7
	[256; 512]	384	**25.5**	**8.0**

Tab. 2.1: Performance of Various VGG Sizes [21]

By increasing the number of layers, it compensates for the loss of the receptive field of larger kernels. For example by stacking two layers of the 3x3 conv. layers we get an effective receptive field of 5x5 and by adding one more layer we get a 7x7 effective receptive field. With this design, we achieve a more discriminative decision function because more non-linear rectification layers will be used. On top of that it drastically reduces the number of parameters (single 7x7 conv. layer requires $49C\hat{2}$ of parameters and three 3x3 conv. layers require only $27C\hat{2}$ of parameters where "C" is the number of channels). [21]

The table 2.1 shows the effect of network size on the accuracy. The larger networks generally perform better than smaller ones.

2.1.5 Inception Netwrork

This architecture was published in 2014 mainly by engineers from Google. The main hallmark of this architecture is the improvement of the architecture without increasing the number of parameters in the network. Contrary, the proposed architecture uses 12 times fewer parameters than AlexNet while being significantly more accurate. The result is a 22-layer architecture with only 5 million parameters. [22]

Previous architectures tried to improve the network by increasing the depth of the network. However, this approach comes with three major drawbacks. Deep networks are prone to overfitting, it is more difficult to learn these networks due to vanishing gradient and they are very computationally expensive. The solution

proposed by Inception architecture is to go not just deeper, but also wider.

As mentioned in the paper, the architecture of this network heavily uses a deep network structure called “Network In Network” [23] which uses a micro neural networks within CNN. The feature maps are obtained by sliding the micro-networks over the input in a similar manner as CNN and the result is then fed into the next layer. In [23] they argue that classical convolutional filter in CNN is a generalized linear model with a low level of abstraction. To achieve higher levels of abstraction, they replace the linear filter with the tiny fully connected neural network (see Fig. 2.6) which is a general nonlinear function approximator whose weights can be learned using back-propagation as well. The aim of both types of filters stays the same - map the local receptive field to an output feature vector. [22]

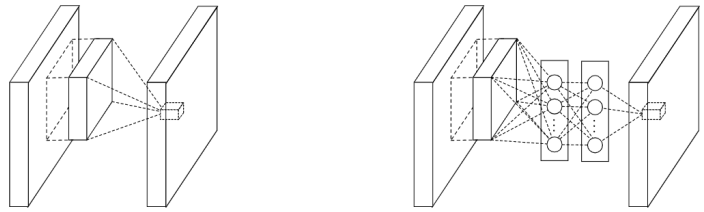


Fig. 2.6: Comparison of Linear Convolution Layer and Mlpconv layer. (The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network) [23]

The new convolutional layer inspired by [23] is called "inception layer" (sometimes called "inception module"). The inception layer performs convolution on input, with 3 different sizes of filters 1x1, 3x3 and 5x5 (see Fig. 2.7). The outputs of individual filters are concatenated and sent to the next layers. To avoid an explosion of the number of parameters they use 1x1 convolution to limit the number of input channels before expensive 3x3 and 5x5 convolutions.[22]

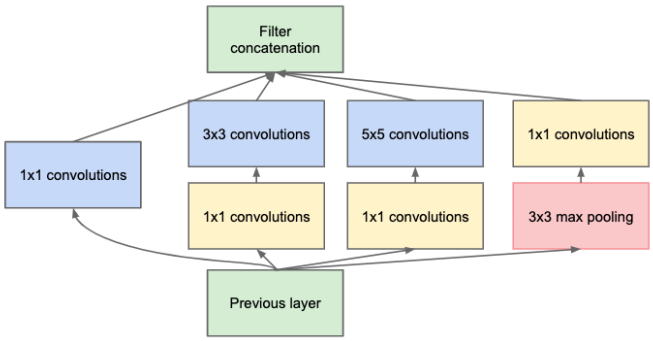


Fig. 2.7: Inception Module [22]

The resulting architecture is 22 layers deep when counting only layers with parameters (27 with pooling layers). The overall number of layers (independent building blocks) used for the construction of the network is about 100. The Fig.2.8 shows that the network has two auxiliary classifiers connected to the intermediate layers. These are used only for training in order to prevent the middle part of the network from "dying out" (vanishing gradient problem).[22]

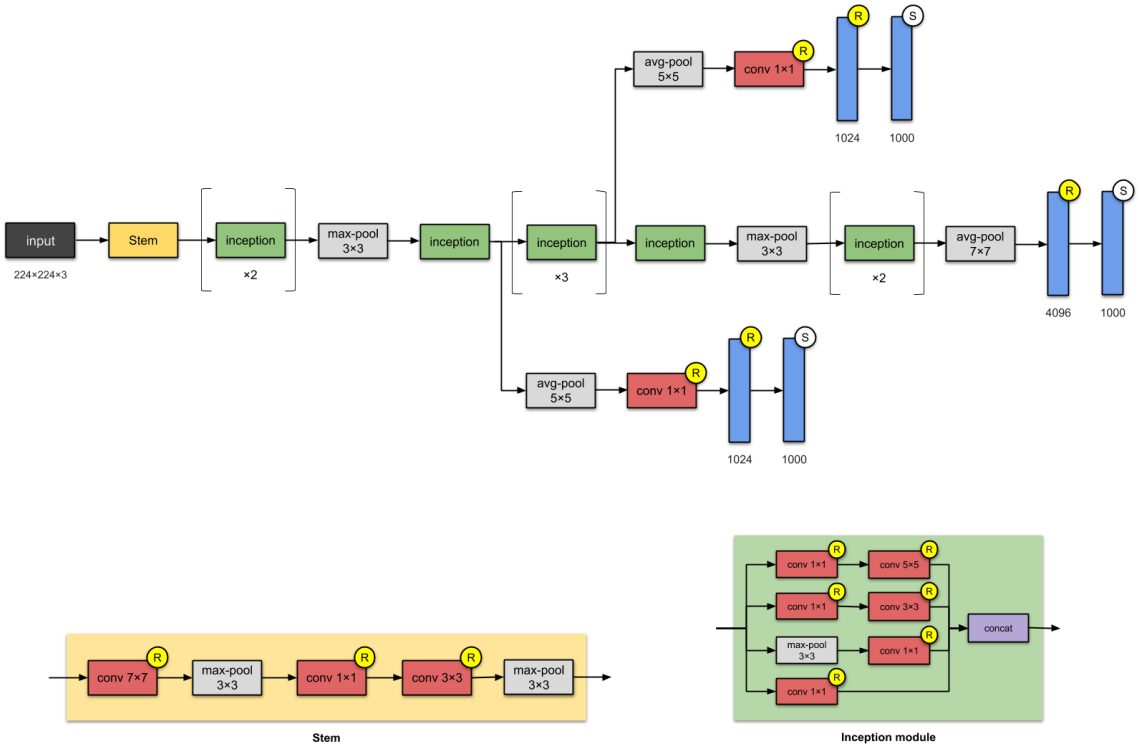


Fig. 2.8: Inception-v1 Architecture [19]

A year later in 2015 were introduced **Inception-v2** and **Inception-v3** in the same paper [24]. Inception-v2 is basically an early prototype of Inception-v3 and is not commonly used. The main goal was to avoid the "representational bottleneck" caused by the drastic reduction of dimensions used in the first version. This was achieved by increasing the computational efficiency by factorization of large convolutional filters (5x5 and 7x7) into a series of 3x3 filters.

For further optimization they factorized the nxn filters into a combination of $1xn$ and $nx1$ convolutions. For example, a 3x3 convolution is equivalent to first performing a 1x3 convolution followed by a 3x1 convolution. They found this method to be **33% cheaper than the single 3x3 convolution**. This leads to a question, whether it would be possible to replace all symmetric convolutional kernels with asymmetric kernels and further improve the computational efficiency. The authors of the paper tried that and they came to the conclusion that asymmetrical kernels

do not work well on early layers, but it gives very good results on medium grid-sizes (mxm feature maps, where m ranges between 12 and 20). The authors also designed new types of inception module that reduces the grid-size while expands the filter banks and removed one of the auxiliary classifiers (because it did not contribute much until near the end of the training process) and used BatchNorm in the remaining auxiliary classifier. The resulting architecture of Inception-v3 has 48 layers with less than 25 million parameters. [24]

2.1.6 ResNet

The aim of this architecture, developed by the Microsoft Research team in 2015, is to address the problem of vanishing gradient that is caused by increasing the number of layers to the CNNs in order to improve the accuracy of the network.

Inspired by "Highway Networks" [25], the ResNet implements a "skip connection" (also called shortcut connections or residuals), that literally skips one or more layers (as shown on Fig.2.9) and adds the output of it to the output of the deeper layer. There are two types of skip connections, one that keeps the input dimensions unchanged, called **identity block** and shortcut that is used when the number of channels and dimensions does not match (typically when the number of channels increases and the size of the feature map decreases) called **conv block** done by 1x1 convolution. [26]

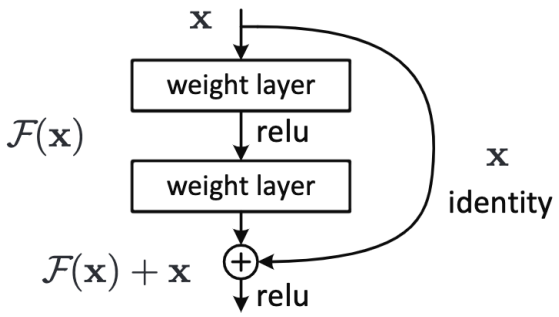


Fig. 2.9: Residual Learning (a building block) [26]

This architecture was applied to networks of different depths. On a shallow network with 18 layers, it had similar Top-1 error as the plain CNN, but it was able to train faster. The expected improvement has shown in networks with 50/101/152-layers. These networks were more accurate than the 34-layer ones by considerable margins. Their most accurate network that won the ILSVRC 2015 had 152 layers with over 60 million parameters. The authors of this network also tried to push the boundaries of their architecture and tried to learn a giant network with **1202 layers**, but they end up with slightly worse higher error than 110 layer network

on CIFAR-10 dataset (error 7.93% and 6.43% respectively). That was most likely caused by overfitting due to an insufficient amount of data.[26]

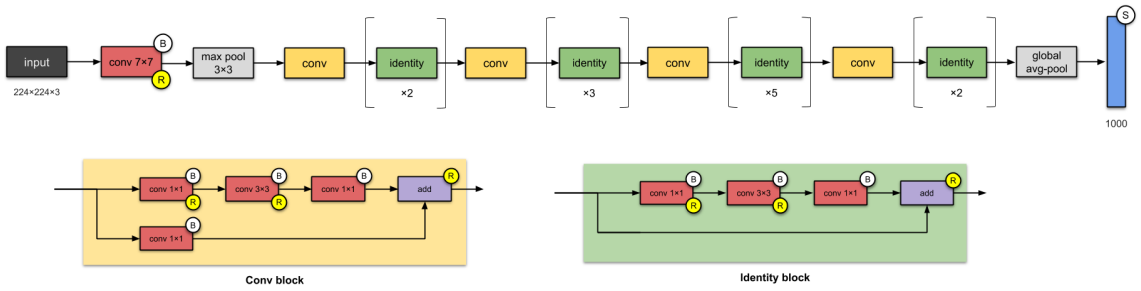


Fig. 2.10: ResNet-50 Architecture [19]

2.2 Object detection

Unlike image classification, object detection is able to localize and classify multiple objects in the image. Even though this task seems very simple for humans, in computer vision it is one of the most fundamental and challenging problems. It forms the basis for high-level vision tasks such as segmentation, scene understanding, object tracking, image captioning, event detection, and activity recognition. Object detection has a wide range of applications ranging from autonomous driving and robot vision to intelligent video surveillance and augmented reality. Originally this task was done by using low-level features such as SIFT and HOG. As you can see in the figure 2.11, the improvement of these methods stagnate around 2010. A new wave of improvements came in 2012 with the arrival of deep learning. [27]

This chapter describes the most common architectures of object detection methods that are based on deep learning.

2.2.1 R-CNN

Region-based Convolutional Network (R-CNN) is one of the pioneering models using successfully CNNs for object detection published in 2014. It is improving the basic object detection algorithms that use inefficient fixed-size sliding window (mentioned in chapter 1.4) with a method called **selective search** developed by J.R.R. Uijlings in 2012 [29]. Selective search initializes small regions in an image and merges them based on the color, texture and size similarity. The R-CNN crops around 2000 of the proposed regions and resizes them to match the input of a CNN which extracts a 4096-dimension vector of features. The features vector is fed into multiple classifiers to produce the class probabilities. Each one of these classes has an SVM classifier

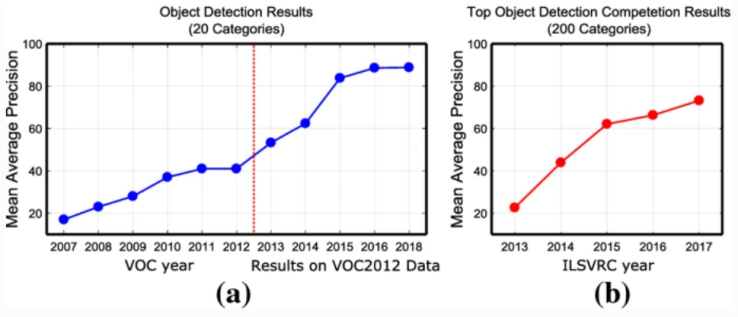


Fig. 2.11: Overview of Recent Object Detection Performance. (we can observe a significant improvement in performance (measured as mean average precision) since the arrival of deep learning in 2012. Image **(a)** Shows results in the VOC2007-2012 competitions. Image **(b)** shows results in ILSVRC2013-2017 competition) [27]

that calculates the probability of class for a given vector of features. The features vector is also fed into a linear regressor to adapt the shapes of the bounding box for a region proposal and thus reduce localization errors. [28]

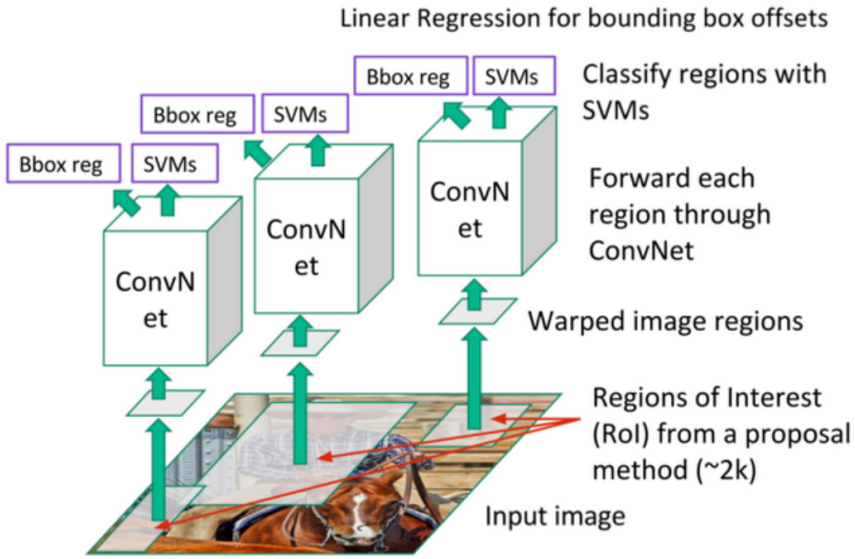


Fig. 2.12: R-CNN Object Detection Pipeline [30]

A year later in 2015 the architecture was improved on its detection speed so they named it **Fast R-CNN**. The speed was improved by two main changes. Firstly, it runs the feature extraction just once over the entire image and propose regions from the generated feature map instead of running the CNN over 2000 overlapping regions. The second significant change is the replacement of many different SVM's to classify each object class with a single softmax layer that outputs the class probabilities directly. That also ease the training process because instead of training SVM for

each class we train just a single network.[31]

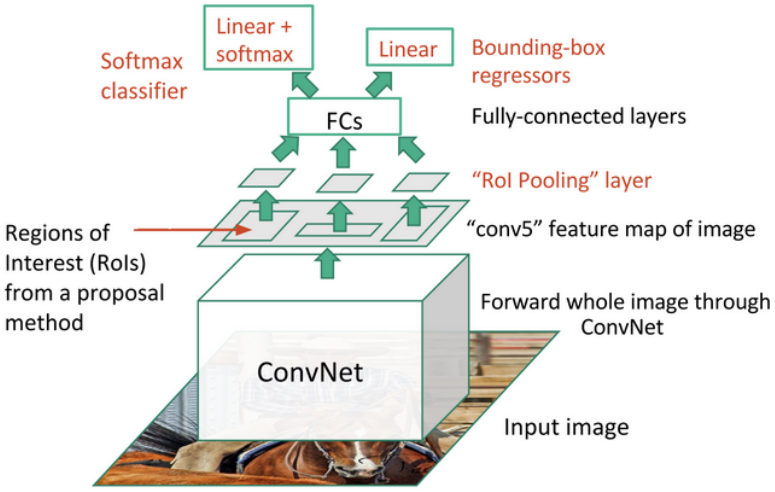


Fig. 2.13: Fast R-CNN Object Detection Pipeline [30]

Another year later in 2016, they published another, even faster version called **Faster R-CNN** removing another speed bottleneck caused by the selective search. They developed a region proposal network (RPN) to directly generate region proposals, predict bounding boxes and detect objects. The input of the RPN is the feature map generated by CNN over the entire image. By sliding a 3x3 window it generates a feature vector that is used to generate region proposals. Each region proposal has 4 coordinates representing the bounding box of the region and "objectness score" (the probability that the bounding box contains an object). Once we have a region proposals, we feed the regions with objectness above a certain threshold level into what is essentially a Fast R-CNN.

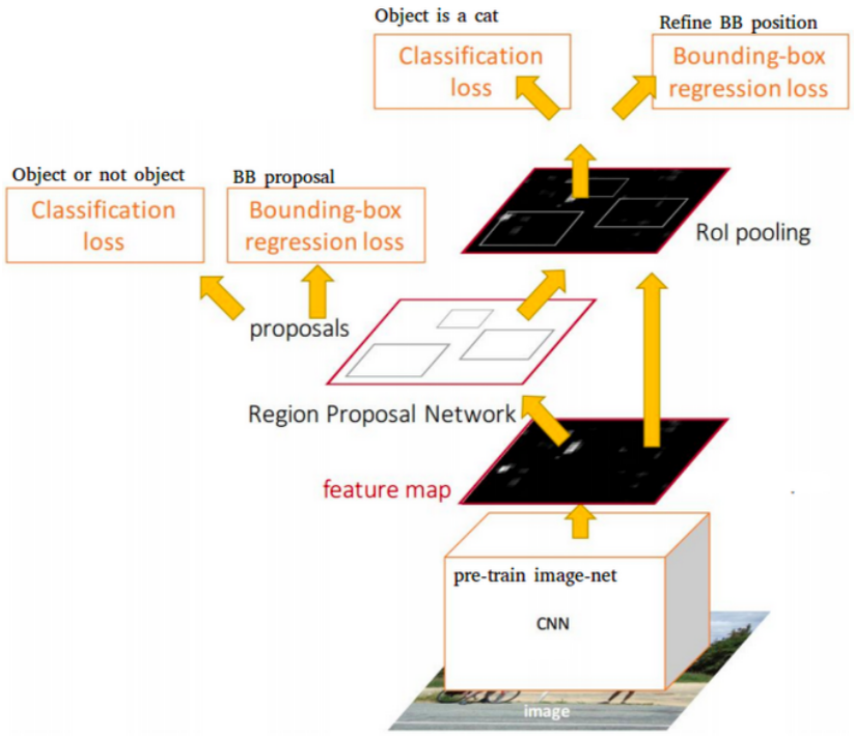


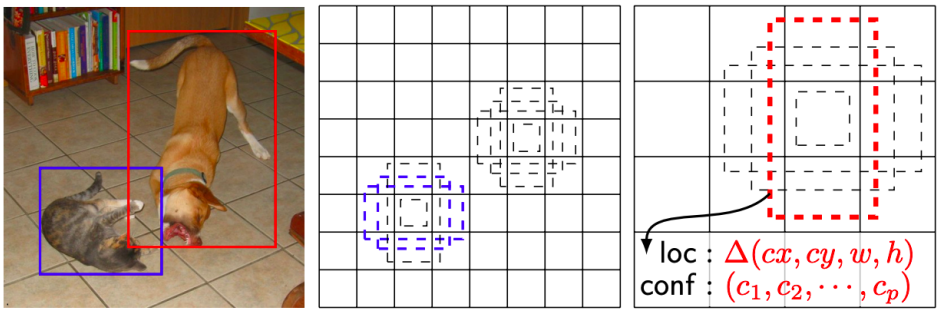
Fig. 2.14: Faster R-CNN Object Detection Pipeline [30]

2.2.2 SSD

Single-Shot Detector (SSD) approaches the problem of localization and classification of objects differently. Instead of splitting the task into two separate steps (region proposal and classification of proposed regions), SSD does it in a "single shot," simultaneously predicting the bounding box and the class as it processes the image.

The SSD architecture is built on top of some feature extracting network (in the Fig. 2.16 it is VGG-16). Instead of the fully connected layers, a set of auxiliary convolutional layers were added in order to get several sets of feature maps at different scales. Afterward, because SSD does not use a region proposal network, it computes both the location and class scores by applying small convolution filters 3x3 to each feature map. For example, in the figure 2.15(b) you can see a 8x8 feature map on which is 3x3 conv. filter applied to compute an offset of 4 anchor bounding boxes $\delta(cx, cy, w, h)$ and class probabilities **for each cell** in the feature map.

Even though it sounds straight forward, there were a lot of challenges to solve. First of all, because there is no region proposal network to filter out boundary boxes with low objectness and it generates several boundary boxes for each cell of each feature map - we get a lot of boundary boxes without any object. To reduce the number of boundary boxes, SSD uses two methods. Firstly, **non-maximum suppression** to group boundary boxes that overlap and most likely contain the



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Fig. 2.15: Detection of Boundary Boxes in SSD [33]

same object. Secondly, the model uses a technique called **hard negative** mining to balance classes during training. In hard negative mining, only a subset of the negative examples with the highest training loss (i.e. false positives) are used at each iteration of training.[33]

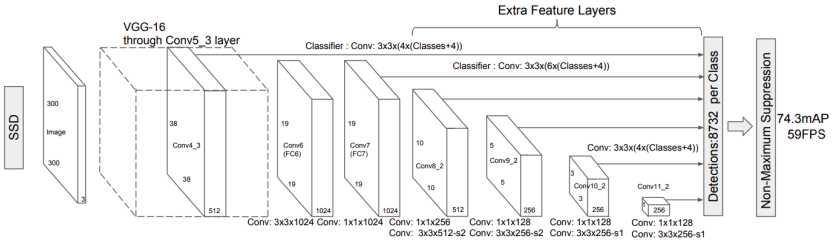


Fig. 2.16: SSD Model Scheme [33]

2.2.3 YOLO

You Only Look Once (YOLO) is an architecture published in the same year as the SSD network working in a similar manner. It predicts bounding boxes and class probabilities with a single network in a single evaluation.

Firstly it uses a CNN inspired by by the GoogLeNet model to calculate the feature map. The final layer outputs a $S * S * (C + B * 5)$ tensor corresponding to the predictions for each cell of the grid. Where S is the size of the feature map, C stands for class probabilities, B is the predicted bounding box and it predicts 5 bounding boxes per cell.

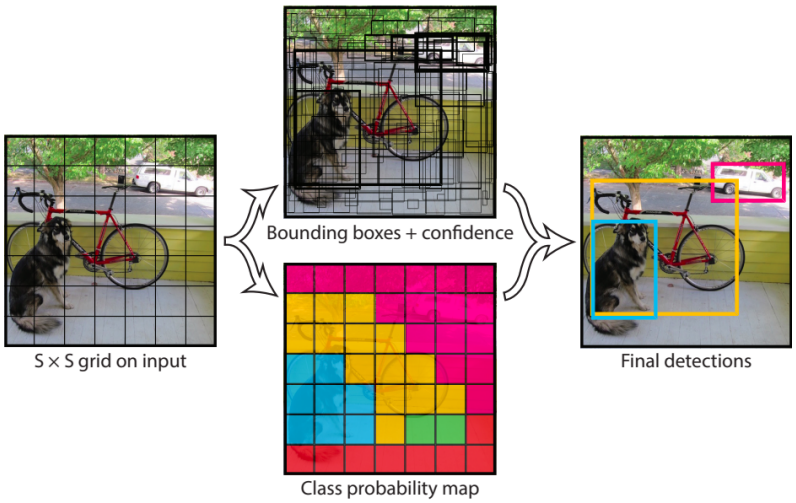


Fig. 2.17: YOLO Object Detection [34]

As you can see in the figure 2.18, the main difference in the architecture from SSD is that the bounding boxes and class probabilities are calculated only from the one feature map generated by the last layer of CNN.

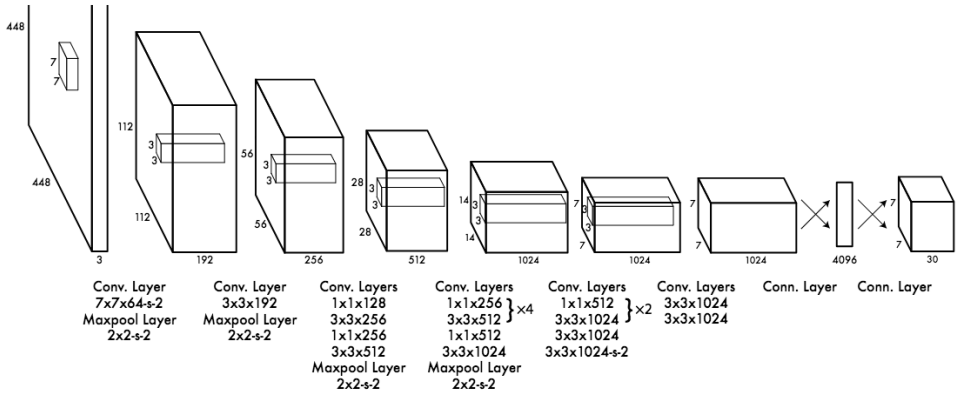


Fig. 2.18: YOLO Architecture [34]

3 Dataset

Dataset is a key component of every machine learning algorithm. To be able to take full advantage of classification and regression algorithms we have to feed the network with a quality dataset. There is no universal definition of a perfect dataset because the requirements depend on the type of the algorithm, architecture, type of task as well as the size of the network. The main indicators of the dataset quality is **size** of dataset, **variation** of data and the **distribution** of objects in classes. The rule of thumb is that the bigger the network, the bigger and more diverse dataset we need to learn a general model. Especially deep learning networks with large capacity are able to learn the model by heart and perform with great accuracy on the training dataset, but quickly lose its accuracy on new data.

Since the object detection algorithms mostly use supervised learning, it is necessary to have annotations. The image annotation is very labor-intensive work that has to be done manually. Luckily, the number of publicly available datasets that are already annotated is increasing in recent years. This chapter will firstly present some commonly used datasets followed by a publicly available, task related datasets of user interfaces. The end of the chapter deals with analysis of the printer screens dataset and possibilities of image prepossessing.

3.1 General Publicly Available Datasets

This sub-chapter will introduce some of the publicly available datasets that are commonly used in computer vision.

3.1.1 MNIST Dataset

The Modified National Institute of Standards and Technology database of hand-written digits is an elementary dataset for computer vision. It has 60k training examples and 10k test examples of hand-written digits for each number, e.i 0-9 which are placed in the center of the 28x28 pixel grayscale image. This dataset became the "Hello World" of machine learning because even though it seems like a challenging task, it is surprisingly simple to obtain accuracy around 90%, even with poorly designed machine learning algorithms. By using a trivial CNN network, it is possible to get an accuracy of around 99%. The dataset is available at [38].

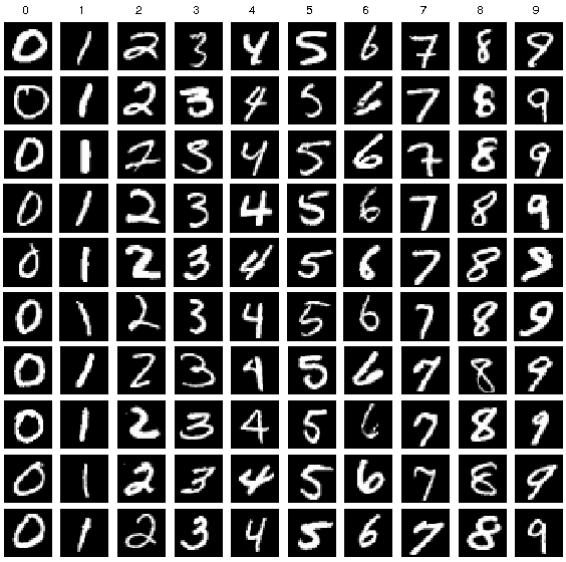


Fig. 3.1: Example of Images in MNIST Dataset

3.1.2 CIFAR-10 and CIFAR-100 Dataset

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset[1] created by Alex Krizhevsky. Both datasets consist of 60k color images formatted in 32x32 pixels, split into a training set (50k images) and testing set (10k images). CIFAR-10 classifies images into 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck) and CIFAR-100 into 100 classes. The best accuracy achieved in "Kaggle Object Detection Competition" is 95% on CIFAR-10 and 72% on CIFAR-100.[40][41] The datasets can be found here [39].

3.1.3 ImageNet

This dataset is organized according to the WordNet hierarchy (see Fig. 3.3). It was created jointly by Standford University and Princeton University for a computer vision competition called The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) where participating teams were challenged with five main tasks: object classification, object localization, object detection, object detection from video and scene recognition. ImageNet dataset consists of around 14 million images in total for nearly 22k different categories of objects.

The main idea of this project is to have 1000+ images for each and every synset (node in WordNet graph) in order to have a visual hierarchy to accompany WordNet.

[1]Dataset link: http://groups.csail.mit.edu/vision/TinyImages/

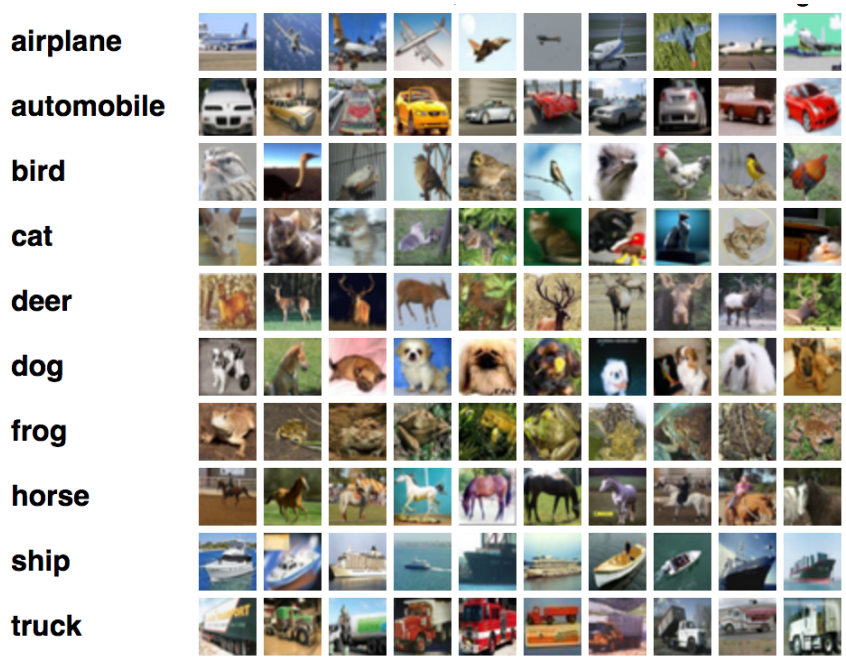


Fig. 3.2: CIFAR-10 Dataset Example

3.1.4 PASCAL VOC Dataset

The Pattern Analysis, Statistical Modeling and Computational Learning Visual Object Classes dataset was open-sourced by a research institute funded by the European Union. A total of 11 530 images are included in this dataset, where each image contains a set of objects, out of 20 different classes, making a total of 27 450 annotated objects. Although the variety of classes and the size of the dataset is much more limited in comparison with ImageNet, PASCAL VOC is more widely used in the early development of object detection and image segmentation.[42]

3.1.5 MS COCO Dataset

Microsoft Common Objects in Context Dataset is a dataset for the Common Objects in Context Challenge which includes five annotation types (object detection, segmentation, person keypoints detection, stuff segmentation, and caption generation). There are about 330k images with over 1.5 million object instances. Even though there are only 91 categories, the number of images per each category is at least 5000 which allows the machine to learn the detailed characteristics of each class. What makes this dataset to stand out is that dataset contains also annotations for **object segmentation** (which is much more difficult than only image classification or drawing boundary box around the object) and **each image has five textual captions** per image such as “a skater jumps over the curb between trees” and “a

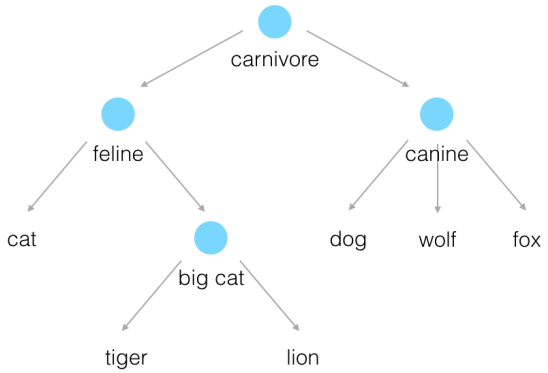


Fig. 3.3: Example of a WordNet Synset Graph

black and white photo of someone on a skateboard". [43] The dataset is available at [44]

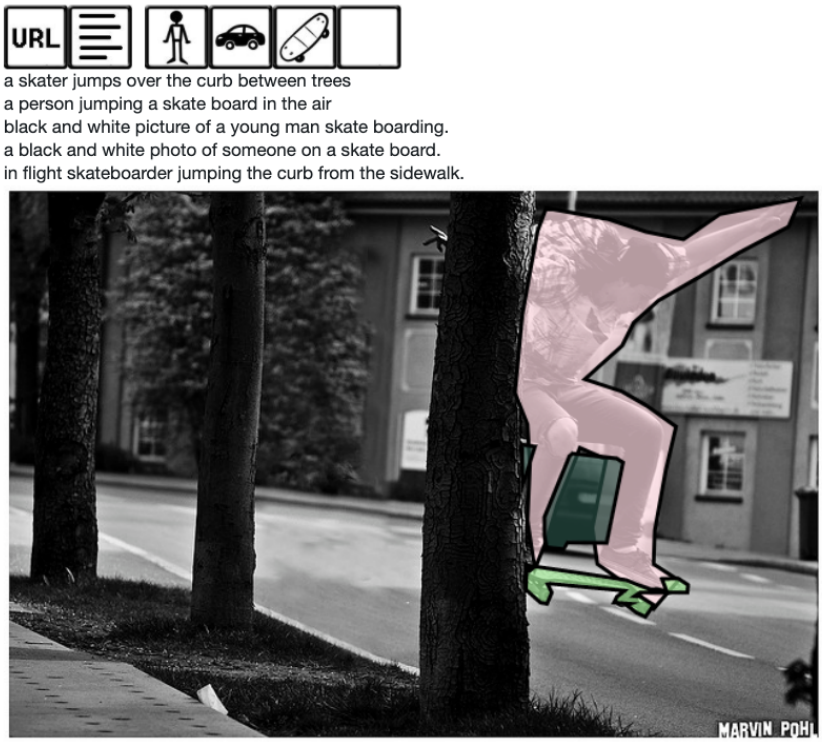


Fig. 3.4: Example of Image in COCO Dataset

3.2 Dataset Creation

This sub-chapter focuses on the analysis of task-related datasets. Firstly, we will analyze publicly available datasets and then the printer dataset that is available for

this task.

3.2.1 Rico

Rico is a mobile app dataset for building data-driven design applications created by the University of Illinois. It has 9.7k Android applications with 72k unique UI screens. The goal of the accompanied project is to create five classes of data-driven applications: design search, UI layout generation, UI code generation, user interaction modeling, and user perception prediction.

Each screenshot is in PNG format and has a detailed view hierarchy in the JSON file (see the listing 3.1). All elements in the UI can be accessed by traversing the view hierarchy starting at the root node and iterating through each "children" property and reading the "class" and "bounds" properties to extract the individual UI elements and their boundary boxes.


```
{
  "activity_name": "com.funforfones.android.chicagocta/
     com.funforfones.android.chicagocta.MainActivity",
  "activity": {
    "root": {
      "scrollable-horizontal": false,
      "draw": true,
      "ancestors": [
        "android.widget.FrameLayout",
        "android.view.ViewGroup",
        "android.view.View",
        "java.lang.Object"
      ],
      "clickable": false,
      "pressed": "not_pressed",
      "focusable": false,
      "long-clickable": false,
      "enabled": true,
      "bounds": [36, 84, 1404, 2392],
      "visibility": "visible",
      "content-desc": [null],
      "rel-bounds": [0, 0, 1368, 2308],
      "focused": false,
      "selected": false,
      "scrollable-vertical": false,
```

```
      "children": [....],
      "adapter-view": false,
      "abs-pos": true,
      "pointer": "2e18ce7",
      "class": "com.android.internal.policy.
         PhoneWindow$DecorView",
      "visible-to-user": true
    },
    "added_fragments": [],
    "active_fragments": []
  },
  "is_keyboard_deployed": true,
  "request_id": "1350"
}
```


Listing 3.1: JSON File With Description of Screenshot

Besides the JSON file with the hierarchical information about the screen layout, each screenshot has a file containing user interaction traces, application metadata (e.g. category, average rating, number of ratings, and number of downloads) and a 64-dimensional vector representation encoded visual layout. This vector was created by Deep AutoEncoder network (see Fig. 3.5) that is differentiating between text and non-text elements. This way it is possible to encode the screenshot into a numerical array and find UIs with similar layout by comparing these vectors.

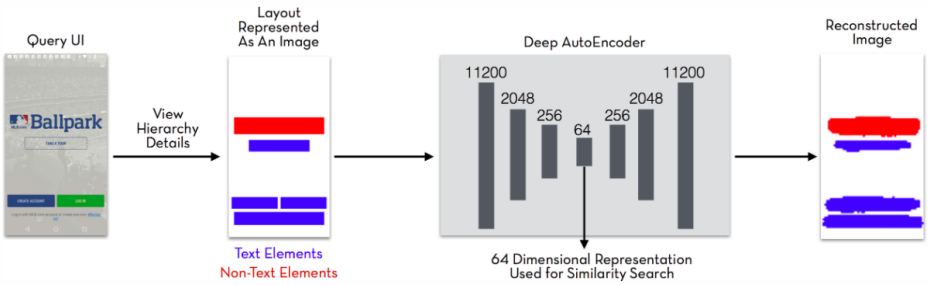


Fig. 3.5: Deep AutoEncoder Used in Rico Project

3.2.2 ReDraw

Another publicly available dataset is ReDraw dataset [47]. This dataset was created in an automated fashion by mining and automatically executing the top 250 Android apps in each category of Google Play excluding game categories, resulting in **14k**

unique screens and **191k labeled GUI-components**. Even though this dataset is smaller than RICO dataset, it is potentially more suitable for my task because it is also used for classification of UI elements so the dataset was cleaned from some "noisy" components.

The project accompanied by this dataset is focused on machine learning based prototyping of smartphone GUIs. The goal is to speed up the prototyping of new applications by automatically converting design muck-ups of the apps created in Photoshop or Sketch into a executable code. The proposed approach consists of three steps. In the first step the UI elements are cropped using the combination of OCR (Optical character recognition) and classical computer vision techniques such as edge detection and contours. The proposed regions with potential UI elements are cropped and classified by a convolutional neural network which is similart to AlexNet with 2 less convolutional layers (3 instead of 5). After classification, they rebuild the UI hierarchy from the classified components using the KNN (K-Nearest Neighbors) algorithm.

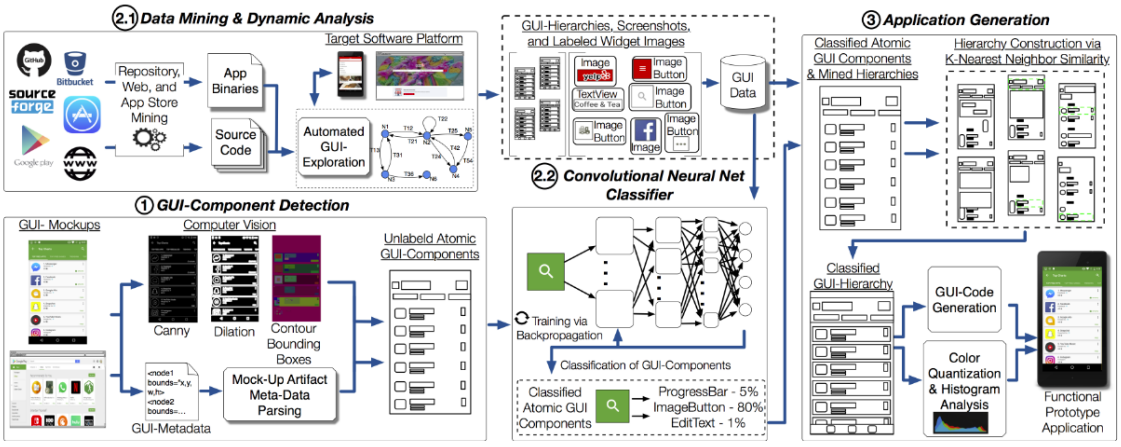


Fig. 3.6: Overview of Automated GUI-Prototiping [48]

The table 3.1 shows, that they are classifying the UI elements into **15 classes**. Like in many datasets produced by “naturally” occurring data, the android dataset suffers from imbalanced classes. The majority of elements are "TextView" (static text) and "ImageView" (static image) represented by tens of thousands of examples followed by "Button", "ImageButton" and "EditText" (input text field) represented by several thousands of examples. This problem was solved in this project by data augmentation. To balance the dataset, two types of data augmentation were performed. Color perturbation that modified the existing images and synthetic app generation which creates completely new screenshots by implementing an app synthesizer capable of generating Android apps consisting of only underrepresented components. This way they ensured, that each class has at least 5000 examples.

GUI-C Type	Total No. (C)	Tr (O)	Tr (O+S)	Valid	Test
TextView	99,200	74,087	74,087	15,236	9,877
ImageView	53,324	39,983	39,983	7,996	5,345
Button	16,007	12,007	12,007	2,400	1,600
ImageButton	8,693	6,521	6,521	1,306	866
EditText	5,643	4,230	5,000	846	567
CheckedTextView	3,424	2,582	5,000	505	337
CheckBox	1,650	1,238	5,000	247	165
RadioButton	1,293	970	5,000	194	129
ProgressBar	406	307	5,000	60	39
SeekBar	405	304	5,000	61	40
NumberPicker	378	283	5,000	57	38
Switch	373	280	5,000	56	37
ToggleButton	265	199	5,000	40	26
RatingBar	219	164	5,000	33	22
Spinner	20	15	5,000	3	2
Total	191,300	143,170	187,598	29,040	19,090

Tab. 3.1: Distribution of UI classes in ReDraw dataset. Abbreviations for column headings: "Total No. (C)"=Total No. of GUI-components in each class after cleaning; "Valid"= Validation; "Tr(O)"= Training Data (Organic Components Only); "Tr(O+S)"= Training Data (Organic + Synthetic Components).[48]

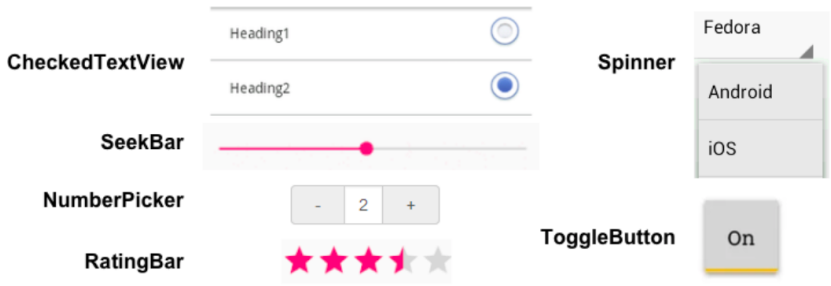


Fig. 3.7: Example of Less Known UI Elements

The confusion matrix in the figure 3.8 shows the percentage of each class on the y-axis, that were classified as components on the x-axis. Thus, the diagonal of matrix (highlighted in blue) corresponds to correct classifications. The overall top-1 precision achieved by the CNN is 91.1%.

	Total	TV	IV	Bt	S	ET	IBt	CTV	PB	RB	TB	CB	Sp	SB	NP	RBt
TV	9877	94%	3%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
IV	5345	5%	93%	1%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Bt	1600	11%	6%	81%	0%	1%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	37	5%	3%	0%	87%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%
ET	567	14%	3%	2%	0%	81%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
IBt	866	4%	23%	1%	0%	0%	72%	0%	0%	0%	0%	0%	0%	0%	0%	0%
CTV	337	7%	0%	0%	0%	0%	0%	93%	0%	0%	0%	0%	0%	0%	0%	0%
PB	41	15%	29%	0%	0%	0%	0%	0%	56%	0%	0%	0%	0%	0%	0%	0%
RB	22	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%
TBt	26	19%	22%	7%	0%	0%	0%	0%	0%	0%	52%	0%	0%	0%	0%	0%
CB	165	12%	7%	0%	0%	1%	0%	0%	0%	0%	0%	81%	0%	0%	0%	0%
Sp	2	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
SB	39	10%	13%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	78%	0%	0%
NP	40	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	95%	0%
RBt	129	4%	3%	2%	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	89%

Fig. 3.8: Confusion Matrix for CNN Abbreviations for Column Headings Representing GUI-component Types: TextView (TV), ImageView (IV), Button (Bt), Switch (S), EditText (ET), ImageButton (IBt), CheckedTextView (CTV), ProgressBar (PB), RatingBar (RB), ToggleButton (TBt), CheckBox (CB), Spinner (Sp), SeekBar (SB), NumberPicker (NP), RadioButton (RBt) [48]

3.2.3 Printer Dataset

The available printer dataset was captured by a The Basler acA1920-40gc GigE camera with the Sony IMX249 CMOS sensor that delivers 42 frames per second at 2.3 MP resolution. The dataset consists of around 1400 images of cropped displays from an image captured by a camera that is above the display (see the camera setup in Fig. 3.9). The image size may vary depending on the device's screen but the vast majority of screens have aspect ratio 16:9 with the resolution of cropped display image around 1280x720px (width, height). Since it is a photo of a screen, most of the images suffer from a subtle moiré pattern which is caused by the fine pattern of pixels on the display that interfere with the pixels in the camera. The example of the image from the dataset can be seen in the figure 3.10.

Class Selection

First of all, it is necessary to decide which UI elements (classes) to detect in the printer dataset. I have decided to start with the list of 15 classes used in the ReDraw dataset and remove classes that are not relevant for the printer dataset or tasks for which might be the detector used. The decisions to keep or remove a class from the list was based on the occurrence of the class in the printer dataset (judgment made

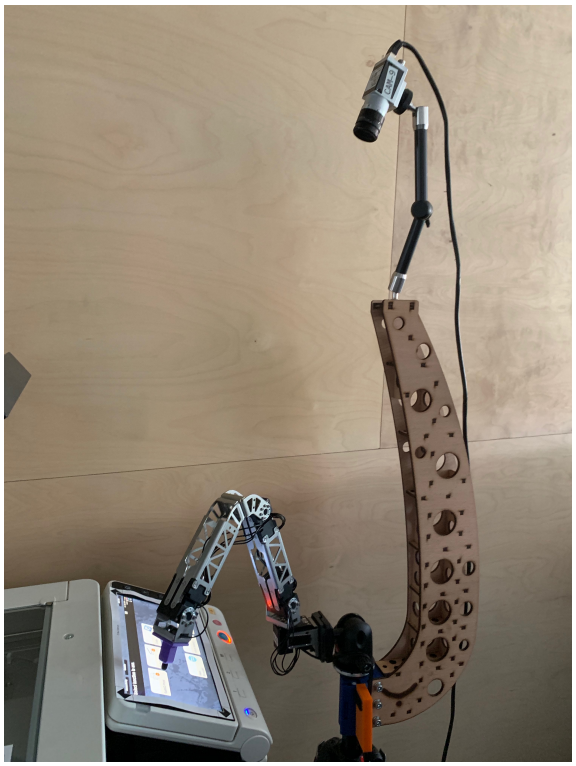


Fig. 3.9: Camera Setup

by skimming through the dataset) and potential utility of the class in the future applications of detection algorithm.

Firstly, I removed SeekBar, ProgressBar, NumberPicker and RatingBar from the list because the elements appear in the printer dataset very rarely. Subsequently, I have decided to remove CheckedTextView because it rather a composition of a text with a checkbox or radio-button than an elementary UI component. Finally, I merged ToggleButton, Spinner, ImageButton and Button into a single class called Button mainly because the appearance and the behavior of these elements is very similar. For example it is usually difficult to judge whether the element is ToggleButton or a Button solely based on the appearance of element. The difference usually stems from the context of current screen. I also considered to differentiate between ImageButton and Button but in the process of annotation I have realized that it is much more difficult to find a line differentiating these two classes than it seems at the first sight so I have decided to merge them into a single class Button. The resulting list of seven classes:

1. StaticText,
2. EditText,
3. Button,
4. RadioButton,

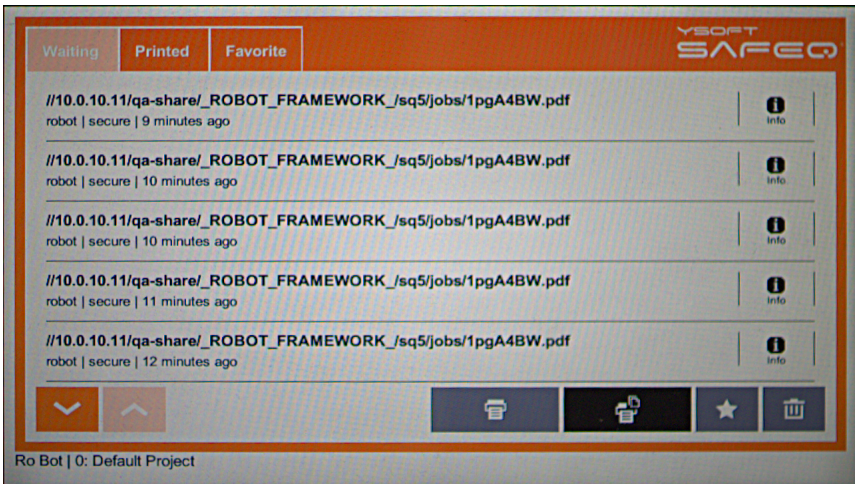


Fig. 3.10: Example of Input Image

5. Switch,
6. CheckBox,
7. StaticImage.

Dataset Annotation

As the dataset was not annotated or anyhow filtered from corrupted images, it was necessary to do it by hand. Object detection tasks require to draw a boundary box (rectangle) around each object in the image and label it with a class name. There is a lot of different image annotation tools that can make this painful job a little bit more acceptable. The most commonly used tools for smaller tasks are LabelImg [49] and VGG Image Annotator [50] which offer just the core functionality. There are also more complex ones such as Labelbox [51] and CVAT (Computer Vision Annotation Tool) [52] which offer more functionality, project management (in case it is possible to split the work between more people) and are compatible with more input and output formats.

I have decided to use CVAT mainly because it might be beneficial for future tasks to use a tool with some project management capabilities. It also multi-platform, open-source, it runs in the browser, it supports keyboard mapping which makes annotation of much faster (especially if you have less than ten classes) and supports multiple annotation formats:

1. CVAT XML v1.1 for images,
2. CVAT XML v1.1 for a video,
3. Pascal VOC,
4. YOLO,
5. MS COCO Object Detection,

6. PNG class mask + instance mask as in Pascal VOC,
7. TFrecord,
8. MOT,
9. LabelMe.

Another advantage is relatively simple installation and scalability since it is a web-app running on a server in Docker and it supports a lot of automation instruments (such as automatic annotation using the TensorFlow Object Detection API and video interpolation). On the other hand it runs only with Chrome and due to many functions the UI is quite complicated and not very intuitive so the setup of an annotation task can be tricky for a first time.

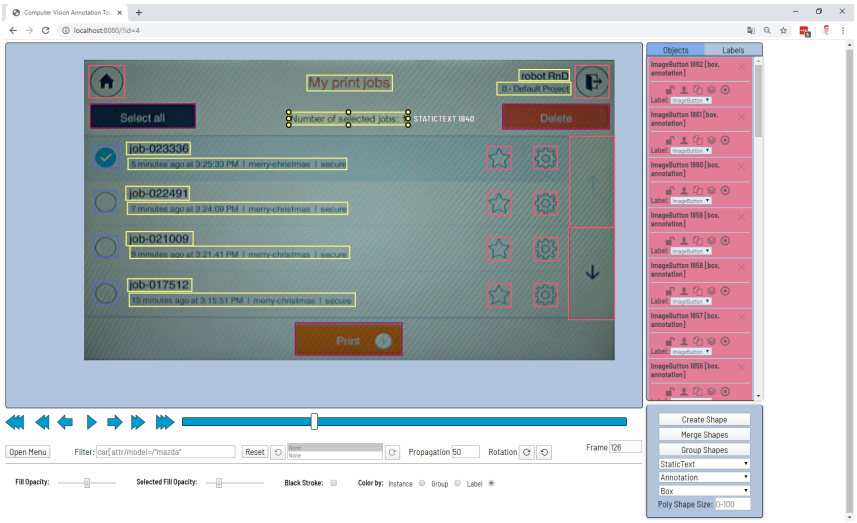


Fig. 3.11: Screenshot of the CVAT Annotation Tool

After the annotation and removing corrupted images we ended up with around 1218 images containing 20 093 objects classified into seven classes. The distribution of classes can be seen in the table 3.1. It is clearly visible that the dataset is biased in a similar way as the Redraw dataset.

In order to get more examples of underrepresented UI elements such as RadioButton, Switch and Checkboxes, we filtered images containing these elements from the ReDraw dataset. Even though the ReDraw dataset comes with XML files containing information about classes and boundary boxes of objects in the image, these data were not reliable so I had to convert those XML files into a PASCAL VOC format, load it into CVAT Annotation tool and correct the wrong annotations by hand. This resulted in 1 257 additional images in the dataset and partially addressed the problem with underrepresented classes. See the distribution of the UI elements in annotated part of Android images from ReDraw dataset in the table 3.2

Class Name	Printer	Android
Button	12084	6584
StaticText	5860	10571
EditText	430	1082
StaticImage	1246	2402
CheckBox	338	1050
Switch	71	424
RadioButton	64	966

Tab. 3.2: Distribution of Elements in the Printer and Android Dataset

3.3 Dataset Preprocessing

Performance of the neural network is dependent mainly on the architecture of the network and the quality of dataset. Therefore dataset preprocessing can make a significant difference in the speed of learning and the overall accuracy of the model. The following chapter describes the most common prepossessing operations.

3.3.1 Dimensions Unification

Since most CNNs work with a fixed size of input data, it is necessary to unify the dimensions of input data before proceeding to the next steps.

Firstly, we have to decide whether to use all channels of the color image or convert it to single-channel grayscale image. There are often considerations to reduce the number of channels, when the neural network performance is allowed to be invariant to that dimension, or to make the training problem more tractable.

We also have to take into account the width and the height of the input image. Some networks were pretrained on various image sizes which give us the possibility to use images with various sizes. If we opt for the unification of all images we can use either cropping or resizing. The cropping is mostly used in case of image classification because the object is usually located in the center of frame but in case of object detection the possibilities might be limited due to the distribution of objects in the frame.

3.3.2 Normalization

Normalization ensures that pixels have a similar data distribution. This makes convergence faster while training the network. Normalization is usually done by subtracting the mean μ from each pixel x and then dividing the result by the standard deviation σ (3.1).

$$z = \frac{x - \mu}{\sigma} \tag{3.1}$$

3.3.3 Data Augmentation

The goal of data augmentation is to artificially produce new data by applying various transformations on original dataset in order to get more data and reduce overfilling while training. The transformations can be in general divided into three categories. Geometry based transformations (rotation, cropping, flipping, etc.), color-based transformations (change of contrast, brightness or modifying color scheme) or adding noise to the images.

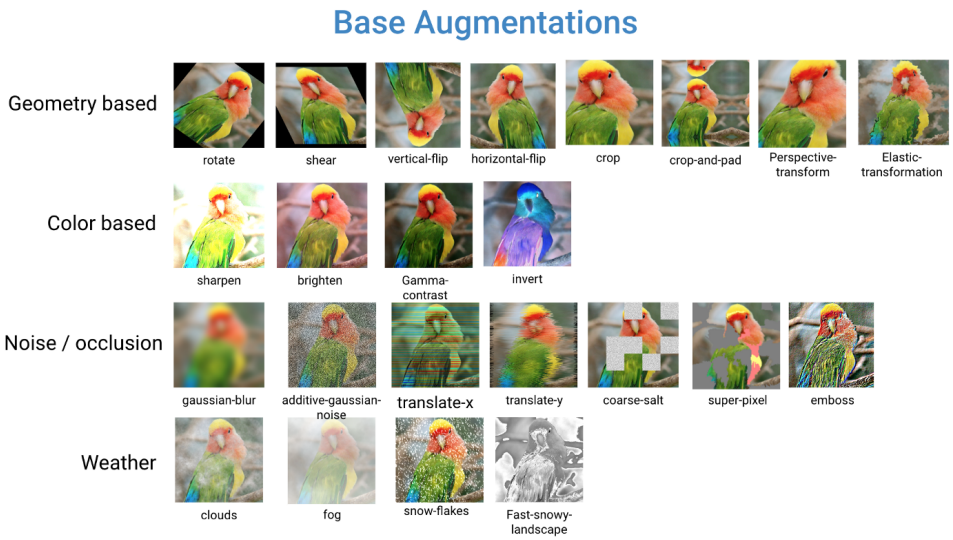


Fig. 3.12: Example of Image Augmentation [55]

3.3.4 Moiré Pattern

Since of the moiré pattern in the images, there arises question whether to suppress it in a prepossessing stage or to keep it there and let the network handle it. In general the image preprocessing for CNNs is very minimalistic (typically resizing and normalization). On the other hand, removal of the moiré pattern can lead to decreasing the problem complexity which leads to faster learning or the possibility to use the smaller and faster network. I have tried to remove the pattern by using several approaches.

First of all, I tried basic methods to filter out noise from the image such as Median Filter, Gaussian Filter and Bilateral Filter but none of them resulted in any significant improvement. They reduced the general noise in the image but they did not affect patterns in the image at all.

Another approach that I have tried is to convert the image to the frequency domain using the DFT (Discrete Fourier Transform). Due to periodic nature of the moiré pattern it should manifest itself as peaks in the frequency spectrum. By finding and filtering out those peaks it should be possible to suppress the pattern. In order to investigate this hypothesis I have created a script that converted the RGB image into grayscale (for simplification) and afterward I used Fast Fourier Transform to convert the image into a frequency domain. The script allows us to manually draw a filter on top of the frequency spectrum image and then filter it. I did several experiments with this script, but I was not able to significantly improve the quality of the images without generating different patterns on the screen (especially in RGB images). This is because the frequency of noise interfere often with the frequency of objects on the image that we want to keep unchanged. By filtering out those peaks we also remove frequencies that are important for the image and generate some other patterns on the image as a side effect.

The example of relatively successful filtering of a grayscale image using this method can be seen in the figure 3.13. The peaks caused by the moire pattern are marked with the red arrows. In this case peaks are easily distinguishable and do not align with horizontal and vertical frequencies. Unluckily this does not apply for most of the images.

The last algorithm that I have tried is Non-local Means Denoising [56] which works on a similar principle as removing noise by capturing photos of the same scene multiple times from the same position and averaging pixels. This algorithm simulates this approach only with one image by finding similar patches within one image and use them to average pixels. The principle is shown in figure 3.14 where blue patches show similar areas that are averaged. This approach results in much greater post-filtering clarity, and less loss of detail in the image compared with local mean algorithms but is very computationally expensive.

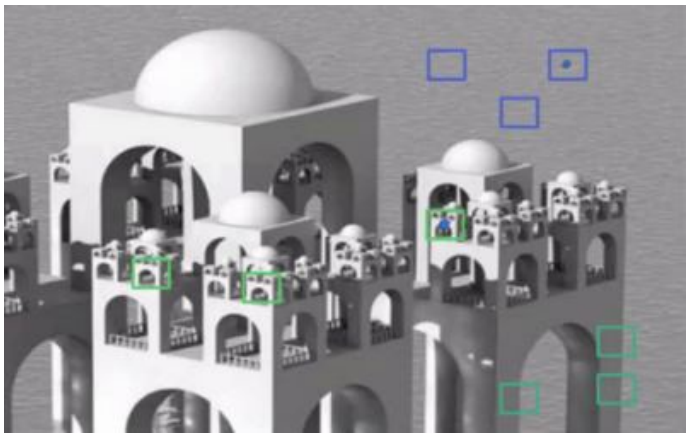
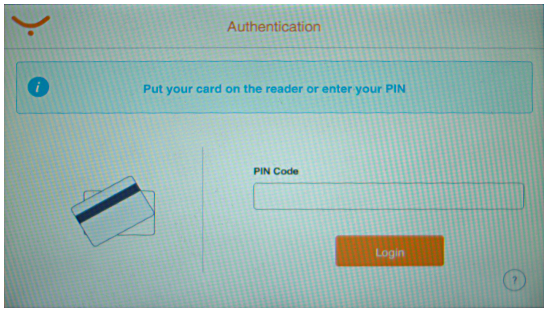
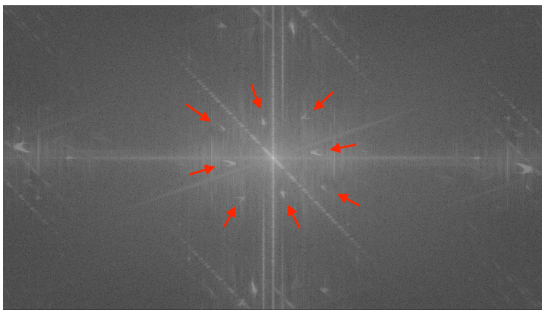


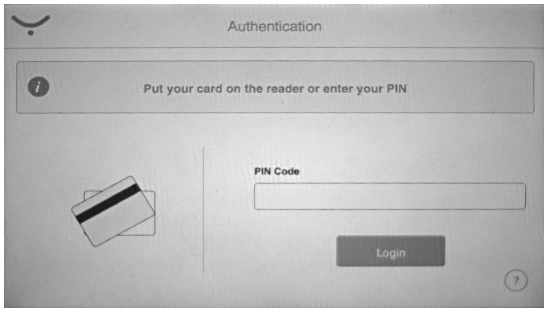
Fig. 3.14: Non-local Means Denoising Principle [53]



(a) Original Image



(b) Frequency Spectrum



(c) Filtered Image

Fig. 3.13: Filtering in the Frequency Domain

The results of Non-local Means Denoising can be seen in the figure 3.15. This method showed the best results from all the tested algorithms. It is not able to completely remove the moiré pattern but it significantly suppresses it, while not affecting the sharpness of image too much.

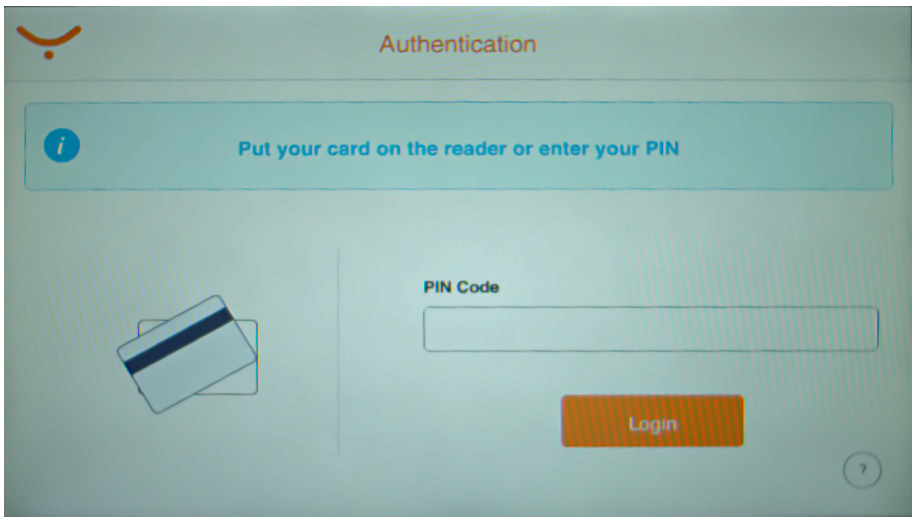


Fig. 3.15: Non-local Means Denoising

4 Implementation

Machine Learning is one of the fastest emerging technologies today. Due to its rising popularity, there is a lot of frameworks that help ML developers to define models in precise, transparent, and concise ways. This chapter will briefly introduce some of frameworks that were considered for this task followed by implementation details of the chosen ML framework.

4.1 Machine Learning Frameworks

4.1.1 ML.NET

ML.NET is Microsoft's open-source, cross-platform, high-level framework for Machine Learning built on top of a Tensorflow C++ API. This framework was my first choice because the rest of system, where my implementation will be deployed, is using the .NET platform. This framework aims to provide easy to use, high level, task-oriented API. It targets different tasks such as Image Classification and Object Detection instead of having more complex API, that could train any kind of deep learning model. The great advantage of this approach is that it allows us to cascade a couple of commands using the Fluent Interface which would translate in hundreds of lines of code in Tensorflow (see Fig. 4.1). The disadvantage of such a high-level APIs is its flexibility. In the case of the ML.NET, it is limited possibility of transfer learning. So far there is a very limited possibility to retrain models. The only possibility is to reuse a part of already pre-trained model and add trainable classifier on top of the network, that will learn to classify objects based on features generated by the pre-trained network. This is also the main reason why I have decided to use some different framework. [57]

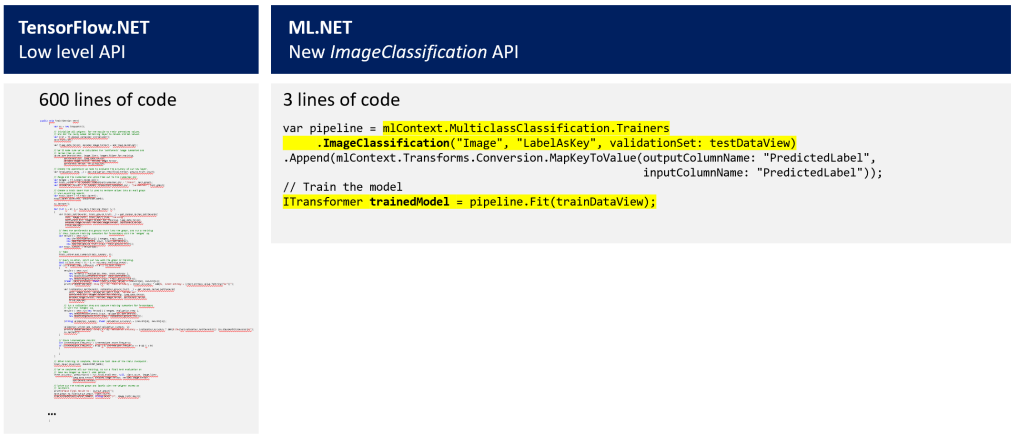


Fig. 4.1: Comparison of ML.NET and Tensorflow Code [57]

4.1.2 Keras

Keras is a high-level neural networks API, written in Python, capable of running on top of TensorFlow, CNTK, or Theano. This framework aims to enable fast prototyping and state-of-the-art research. The library has many models for image classification trained on ImageNet which can be modified and reused for further training on task-specific datasets. Unfortunately, it does not offer any official pre-trained object detection models. The only option is to use community-based models which is not an optimal solution in case you want to experiment with multiple architectures.

4.1.3 Tensorflow Object Detection API

The TensorFlow Object Detection API (TFOD), written in Python, is an open-source framework built on top of TensorFlow that makes it easy to construct, train, and deploy object detection models. The advantage of this framework is its focus only on object detection which results in a wide range of pre-trained models on various datasets [60], a simple modification of training parameters in a configuration file and useful tools for training and evaluation of the model. The disadvantage of this framework is that it is in "research mode" which means that there is nearly no documentation available. Furthermore, some of the errors are solved by modification of source code and the framework is very sensitive to a particular version of dependent libraries. Nevertheless, this framework seems to be a good fit for my task so I have decided to use it.

TFOD Installation

The first step is to install the TensorFlow itself. There are two versions of the Tensorflow. One version using only CPU and second version using GPU. The latter is preferred since most of the machine learning models are optimized for GPUs that makes the training much faster. There is a lot of constraints that we have to fulfill in order to correctly install the version supporting GPU. First of all, we have to use supported graphic cards from NVIDIA card with CUDA Compute Capability 3.5 or higher (CUDA is a parallel computing platform and programming model developed by NVIDIA). The list of compatible NVIDIA graphics and their Compute Capability can be found here [61]. The graphic card that I have available for the training is NVIDIA Geforce RTX 2070 SUPER which has Compute Capability 7.5. Another very important requirement is to make sure that the version of python, TensorFlow, cuDNN, and CUDA is compatible (see the compatibility table in Fig. 4.2). In my

case, I have decided to use Python 3.7.6, Tensorflow-gpu 1.14.0, cuDNN 7.4, and CUDA 10.

GPU

Version	Python version	Compiler	Build tools	cuDNN	CUDA
tensorflow_gpu-2.0.0	3.5-3.7	MSVC 2017	Bazel 0.26.1	7.4	10
tensorflow_gpu-1.14.0	3.5-3.7	MSVC 2017	Bazel 0.24.1-0.25.2	7.4	10
tensorflow_gpu-1.13.0	3.5-3.7	MSVC 2015 update 3	Bazel 0.19.0-0.21.0	7.4	10
tensorflow_gpu-1.12.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.11.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.10.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9

Fig. 4.2: Tested Build Configurations [62]

It is a good practice to create a virtual environment with the required Python version for each project. This can be easily done using the command:

```
conda create -n tf_gpu_env python=3.7.6
```

Then, we can activate the virtual environment and install the TensorFlow using the *pip* tool:

```
conda activate tf_gpu_env
pip install tensorflow-gpu==1.14
```

Now, we have to download and install the supported version of cuDNN and CUDA from the official NVIDIA DEVELOPER website and add paths to it to the *PATH* environmental variable. With the default installation settings, the path should be similar to:

```
SET PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\
   CUDA\v10.1\bin;%PATH%
SET PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\
   CUDA\v10.1\extras\CUPTI\libx64;%PATH%
SET PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\
   CUDA\v10.1\include;%PATH%
SET PATH=C:\tools\cuda\bin;%PATH%
```

Finally, we can install the Tensorflow Object Detection API dependencies using the *pip*:

```
pip install Cython
pip install contextlib2
pip install pillow
```

```
pip install lxml
pip install jupyter
```

and clone the TFOD repository:

```
git clone https://github.com/tensorflow/models.git
```

The TFOD uses COCO object detection metrics for model evaluation during the training which is provided by a cocoapi package [63]. Unfortunately, the official distribution does not support Windows so we have to install an unofficial, modified version that works on Windows from a former Microsoft developer Phil Ferriere from his GitHub [64]:

```
pip install "git+https://github.com/philferriere/cocoapi.
   git#egg=pycocotools&subdirectory=PythonAPI"
```

Because TFOD API uses Protobuf files to configure model and training parameters, it is necessary to compile them first by running the following command from the $< path\_to\_cloned\_models\_dir > /research/$ directory:

```
protoc object_detection/protos/*.proto --python_out=.
```

The penultimate step is to add the installed libraries to PYTHONPATH:

```
SET PYTHONPATH=<path_to_cloned_models_dir>;<
   path_to_cloned_models_dir>\research;<
   path_to_cloned_models_dir>\research\slim
```

It is possible to test whether the installation was successful by running the following script without error:

```
python <path_to_cloned_models_dir>/research/
   object_detection/builders/model_builder_test.py
```

4.2 Training

Before we can start the training process, we have to convert the training data into a supported format and configure the training process.

First of all, we have to split our dataset into a training and evaluation dataset that will be used during the training to measure the performance of the model.

Afterward, we can convert these datasets, which consist of image and XML file with annotations, into a format that is supported by TFOD API called TFRecords. The TFRecord format is a simple format for storing a sequence of binary records. The data can be easily converted from the *Pascal VOC* format into a TFRecords by using a converter that I have written called *convert_pascal_voc_to_tfrecords.py* (see chapter 4.3.1 for more details).

In the following step, we create a label map that maps the object ID number to the object name. The label map has to be in a *pbtxt* (Protocol buffer) format which is a Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. The label map file for my task is:

```
item {id: 1
    name: 'StaticText'}
item {id: 2
    name: 'EditText'}
item {id: 3
    name: 'RadioButton'}
item {id: 4
    name: 'Switch'}
item {id: 5
    name: 'CheckBox'}
item {id: 6
    name: 'Button'}
item {id: 7
    name: 'StaticImage'}
```

Finally, we can download one of the pre-trained models form the model zoo [60]. Each model comes with the "pipeline.config" where we can easily configure the whole training process. This file offers tens of parameters that can be tuned for example batch size, learning rate (static or dynamic), regularizer parameters, the resizer of the input image, image augmentation, number of training steps and many more parameters (the whole configuration file can be found in appendices A). All of the parameters are pre-set with values that were used for model pre-training. The only parameters that we have to change are the number of classes, path to the pre-trained model (that is downloaded from the model zoo), path to label map, path to training dataset and path to testing dataset:

```
model {
  ssd {
    num_classes: 7
```

```
...

  fine_tune_checkpoint: "<...>/model.ckpt>"

...

train_input_reader {
  label_map_path: "<...>/labelmap.pbtxt"
  tf_record_input_reader {
    input_path: "<...>/train.record"
  }
}

...

eval_input_reader {
  label_map_path: "<...>/labelmap.pbtxt"
  tf_record_input_reader {
    input_path: "<...>/test.record"
  }
}
```

After setting of the training parameters in the configuration file, we can finally run the training script *model_main.py* that can be found in the directory *models/research/object_detection*:

```
python model_main.py
   --model_dir=<path to output directory>
   --pipeline_config_path=<path to config file>"
```

where the $--$ *model_dir* is the path to the directory where all training output files will be saved and $--$ *pipeline_config_path* is a path to the config file.

Since TensorFlow saves only five newest models without any option to save the best performing models or all models, I wrote a script *backup_model_files.py* 4.3.2 that periodically checks the *model_dir* folder and backups the saved models:

```
python backup_model_files.py -src=<path-to-model_dir> -
   dst=<path-to-backup-dir>
```

The great advantage of Tensorflow is that it comes with the TensorBoard tool which allows us to monitor the training process. It can be started by starting the TensorBoard server by running the following command in the *model_dir* folder:

```
tensorboard --logdir=<path-to-model-dir> --host=127.0.0.1
```

Then, we can open the TensorBoard dashboard in the web browser using the address `http://127.0.0.1:6006/`. Besides training and evaluation loss, TFOD generates trends of all coco metrics, learning rate as well as few images with detected objects.

The output model file in *model_dir* folder is not actually model that can be used for the detection right away but it is a series of three *"checkpoint"* files e.g.

- model.ckpt-22285.data-00000-of-00001,
- model.ckpt-22285.index,
- model.ckpt-22285.meta.

Checkpoints capture the exact value of all parameters used by a model but it does not contain any description of the computation defined by the model. So we have to convert it into a protobuf file usually called $frozen\_inference\_graph.pb$ which contains the graph definition as well as the weights of the model. The conversion from checkpoint to protobuf file can be done by $export\_inference\_graph.py$ that can be found in $.../models/research/object\_detection$ folder:

```
python export_inference_graph.py
    --input_type image_tensor
    --pipeline_config_path <path to pipeline.config>
    --trained_checkpoint_prefix <path to model.ckpt>
    --output_directory <output-dir-path>
```

4.3 Custom Created Scripts

I have written a small utility module that contains scripts that automate some tasks such as dataset conversion and model evaluation, visualize the detection results, and dataset exploration scripts. All of those scripts can be found in the *utilities* folder that is located in the root directory of the project. All of the utilities are not meant to be run from the command line, except the utility $backup\_model\_files.py$. The input arguments are set directly at the end of the script for example:

```
...

if __name__ == '__main__':
```

```
PATH_TO_ANNOTATION_DIR = r"<path to annotation dir>"

class_distribution_dict = run(annotatiton_dir_path=
    PATH_TO_ANNOTATION_DIR)
print(f'Distribution of classes in the dataset:\n{
    class_distribution_dict}')
```

Even though this may lead to worse code quality, it brings better code flexibility and faster prototyping. On top of that, the vast majority of scripts are not dependent on other scripts so the code quality is not as important.

There are a lot of small scripts such as *compute_class_distribution.py* that iterates thought the annotation files and returns number of objects in each class, *remove_corrupted_images.py* that removes images that were marked as corrupted during the annotation, *remove_black_bars.py* that removes the navigation and notification bar from the android images and many more. This sub-chapter will discuss in detail some of the utility scripts that were used.

4.3.1 Dataset Format Converter

The PASCAL VOC annotation format which is an output of the annotation process has to be converted into a TFRecord format. TFRecord file consists of sequence of *tf.Example* which is a flexible message type that represents a "string": tf.train.Feature mapping.

This can be done using *convert_pascal_voc_to_tfrecords.py* which is a conversion script that has four input parameters:

1. path to the directory with dataset images,
2. path to the label map,
3. path to the directory with annotation files ,
4. name of the output record file.

In the first step, the script finds for every image its belonging XML file and parses the data. The example of the XML annotation file in the PASCAL format can be found in appendix B. Afterward, it converts the image into a byte stream and serializes it together with information about image, boundary boxes and classes into a *tf.Example* message in the following format:

```
tf_example = tf.train.Example(
  features=tf.train.Features(feature={
  'image/height': dataset_util.int64_feature(height),
  'image/width': dataset_util.int64_feature(width),
  'image/filename': dataset_util.bytes_feature(filename),
  'image/encoded': dataset_util.bytes_feature(encoded_jpg),}))
```

This is repeated for every pair of image and XML file in the dataset and serialized into a single *record* file.

4.3.2 Periodic Model Backup

Tensorflow Object Detection API saves only five newest model checkpoints which is not optimal when the model gets overtrained becasuse we would end up with five overtrained models. In order to get the optimally trained model we would have to set a different number of training steps and train the model from the beginning. To avoid that, I wrote a script *periodically_backup_model_files.py*, that periodically back ups all model checkpoints. The script is run from a separate command line during the training:

```
python backup_model_files.py
    -src=<path to model_dir>
    -dst=<path to backup folder>
```

where the *-src* is a training output folder path and *-dst* is the destination folder where will be the model checkpoints copied. The script gets all files from both directories with the following name pattern *'model.ckpt-*'* where the asterisk character is a placeholder for the step number at which was the model saved (e.g. model.ckpt-13463.meta). The script copies the file from *src* directory to *dst* directory in two cases:

1. The file found in *src* directory is not present in *dst* directory.
2. The file is present in both *src* and *dst* directory but have different size.

4.3.3 Model Evaluation

Evaluation of the trained models is done using the script *evaluate.py* with the following input parameters:

1. *model_dir_path* - path to the folder containing the trained model checkpoint and configuration file,
2. $eval\_tfrecord\_path$ - path to evaluated data in the *tfrecord* format ,
3. *eval_img_dir_path* - path to the folder containing evaluated images,
4. *labelmap_path* - path to the labelmap.

The output of script is saved into a $evaluationfolder$ that will be automatically created in the *model_dir_path* folder. The evaluation output contains the full coco evaluation metric, precision, and recall calculated for each class, confusion matrix and it draws the detected results side by side next to the image with the ground truth bounary boxes.

The script consists of several steps. Firstly, it converts model checkpoint into *frozen inference graph* which is a "ready to use" format of the model. It contains both graph definition and trained parameters that can not be trained anymore but can be used directly without any other file. The output *"frozen_inference_graph.pb"* is saved into a *model_dir_path*.

Secondly, we use the converted inference graph and make the predictions on the *eval_tfrecord_path*. The predictions are saved in the *infer_detections.record* file that is also saved in the *evalutionfolder*. Before running the model conversion and the inference on evaluation data, the script checks whether these files already exist and if they do, it skips the first two steps.

Afterward, we parse the *infer_detections.record* file and use the parsed data to calculate the confusion matrix followed by a precision & recall calculation for each class (explained in chapter 1.5.2). The confusion matrix is calculated from detection boxes whose detection scores are greater or equal than 0.5 and the match between the ground-truth box and detected box is considered for IoU greater or equal than 0.5. Then, the duplicates are deleted so each ground-truth box has only one detection box and vice versa. The confusion matrix is updated according to the remaining matches.

Rows in the output confusion matrix represent the target values and the columns represent the predicted values. Besides columns and rows for each class, there is one extra column and row called *None* which is used to indicate when an object of a specific class was not detected, or an object that was detected wasn't part of the ground-truth. Precision and recall for each class are then calculated from the confusion matrix using:

```
for id in range(len(categories)):
  total_target = np.sum(confusion_matrix[id, :])
  total_predicted = np.sum(confusion_matrix[:, id])

  precision = float(confusion_matrix[id, id] / total_predicted)
  recall = float(confusion_matrix[id, id] / total_target)
```

Where *"id"* is index of row and column for a given object category. The output confusion matrix and per class evaluation is saved to *confusion_matrix.csv* and *class_evaluation.csv* files located in *evaluation* directory.[65]

In the next step, the COCO API is used to calculate the coco evaluation metrics. The output metric is saved into a *coco_evaluation.txt* file in *evaluation directory*.

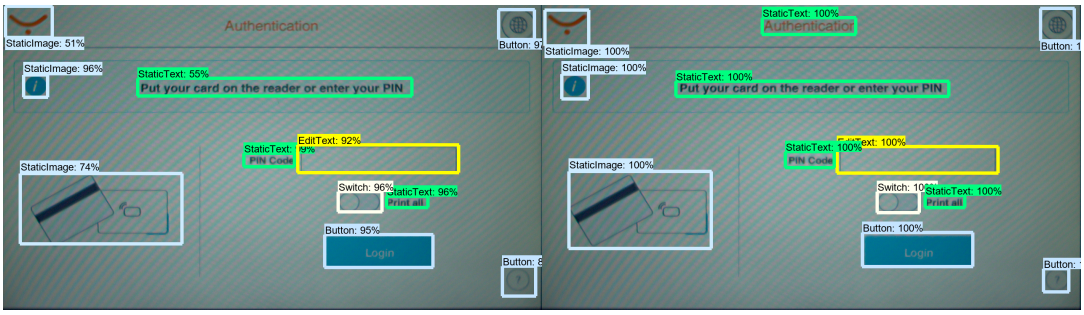


Fig. 4.3: Visualization of Detected Boxes (image on left) and Ground-Truth Boxes(image on right) with Confidence Scores

In the last step, the ground-truth boxes and detected boxes are drawn side by side in the image and saved into a *evaluation/images* folder. The percentage value next to the class name in the image with detected boxes is a confidence of the prediction (see Fig.4.3).

4.3.4 Others

This section briefly mentions the rest of the scripts that were used:

- **bbox_drawer.py** - this file defines a *BboxDrawer* class that is used to draw boundary boxes in the image
- **draw_annotations.py** - gets the path to folder with images and annotation files in Pascal VOC format and draws boundary boxes
- **add_noise.py** - adds various types of noises to the images in the input folder and saves them to output folder
- **compute_class_distribution.py** - loads all annotation files in Pascal VOC format from the input folder and calculates distribution of classes
- **demoire_notebook.ipynb** - Jupyter notebook with various filters whose parameters can be interactively tuned using the Jupyter widgets
- **remove_black_bars.py** - removes notification and navigation bar from the android images
- **remove_class_from_dataset.py** - used to remove e.g. "ListItem" class from annotation files of Android dataset
- **remove_corrupted_images.py** - the image annotation tool does not allow to directly delete corrupted image from dataset so I marked corrupted images with special class called "DELETE_CLASS". When this script finds this class in an annotation file, it deletes the accompanied image as well as the annotation file
- **rename_class_in_xml.py** - used to rename classes in all annotation files located in the input directory. For example, it allowed me to quickly merge

the "ImageButton" and "Button" classes just by renaming "ImageButton" class to "Button".

- **split_annotations.py** - used to split dataset into testing and training datasets
- **coco_utils.py** - wrapper of COCO API. Used to convert annotation files to "ground-truth coco data" and and result of the detections performed by model to "detection coco data" supported by COCO API.
- **confusion_matrix.py** used to calculate confusion matrix and precision & recall for each class
- **convert_pascal_voc_to_tfrecords.py** - generates TFRecord file from from the annotations
- **detection_parser.py** - defines a *DetectionParser* class that is used to parse the output of the model predictions
- **general_utils.py** - set of commonly used functions
- **image_size_distribution.py** - returns a distribution of image sizes in the input folder
- **tf_utils.py** - wrapper of object detection method from TFOD API
- **bbox.py** - object to store boundary box data
- **prediction.py** - object to store predictions

5 Training Process

The training process is divided into two stages. The goal of the first stage is to select a couple of models out of 38 different models available in the Tensorflow detection model zoo [60] by training them on a small dataset subset. In the second stage, we fully train the selected models with various training parameters and discuss how does it affect the training results.

5.1 Selection of Best Performing Models

Since it would be very time consuming to even partially train all 38 available models from the model zoo, I have decided to pre-select 10 models that will be trained. The selection of models was based on the model size, detection speed and architecture diversity. By architecture diversity is meant that it does not make sense to select e.g. *ssd_mobilenet_v2_quantized_coco* and *ssd_mobilenet_v2_coco* because both models are very similar.

Afterward, the selected models were trained on a subset of the printer dataset (see chapter 3.2.3) to decide which models work best with the printer dataset. The decision was made based on the $mAP^{IoU=.50}$ of the models trained and evaluated on the subset of printer dataset as well as on the model speed and $mAP^{IoU=.50}$ on the COCO dataset from the documentation [60]. To get the mAP on printer dataset I have trained all the models from the table 5.1 on 50% of the printer dataset without any preprocessing for only 2850 steps with the same input image size 640x360 and the batch size 8 if possible (the models No.5 and No.6 did not fit into memory so I was forced to use batch size 1). The rest of the training parameters were unchanged.

No.	**Model Name**	$mAP^{IoU=.50}_{printer}$	$mAP^{IoU=.50}_{coco}$	Speed (ms)
1	ssd_inception_v2	0.38	0.24	42
2	ssd_mobilenet_v2	0.34	0.22	31
3	ssd_resnet_50_fpn	0.04	0.35	76
4	ssdlite_mobilenet_v2	0.13	0.22	27
5	faster_rcnn_inception_v2	0.26	0.28	58
6	faster_rcnn_nas_lowproposals	0.04	-	540
7	faster_rcnn_resnet50	0.48	0.30	89
8	faster_rcnn_resnet101	0.35	0.32	106
9	rfcn_resnet101	0.48	0.30	92
10	ssd_mobilenet_v1_ppn	0.00	0.20	26

Tab. 5.1: Comparison of Selected Models

As you can see in the table 5.1, most of the networks were able to achieve over $0.3mAP$, besides the networks No.3, No.6 and No.10 that performed very poorly. Even though the results of this short training are not most likely a good representation of the model's abilities. It is mainly used as a piece of extra information that I can consider when predicting which model might work best with the printer dataset.

Due to the training HW restrictions and the requirement for model speed I have decided to discard models that are too big and slow such as those based on ResNet101 and NAS. From the remaining networks I have decided to select the best performing model using SSD architecture (see chapter 2.2.2) and the best performing model using *Faster R-CNN* architecture (see chapter 2.2.1):

1. model No.1: *SSD Inception v2*,
2. model No.7: *Faster R-CNN Resnet50*.

Since the detection speed is one of the task requirements, I have also selected two models solely on their detection speed:

1. model No.10: *SSD Mobilenet v1 PPN*,
2. model No.4: *SSD Lite Mobilenet v2*.

5.2 Models Tuning

There is no heuristic way how to find the optimal hyperparameters. There are methods such as Grid search and Random search using exhaustive search but these approaches are not very applicable for large neural networks. In my case, I have decided to tune the parameters by conducting several experiments and evaluate the effect of their change on the model performance.

5.2.1 Datasets

For the experiments, I have created three datasets:

1. **Dataset No.1:** Printer dataset without any prepossessing (except resizing),
2. **Dataset No.2:** Printer dataset extended by Android dataset without any prepossessing (except resizing),
3. **Dataset No.3:** Preprocessed printer dataset (denoised + moiré pattern was suppressed) extended by preprocessed Android dataset (blurred with added noise).

Android images used in dataset No.2 and No.3 come from the ReDraw dataset (see chapter 3.2.2). Besides resizing, I removed notification and navigation bar from the Android screenshots and I have added a little bit of random noise and blur to make look more similar to the printer images. The resulting datasets are divided into a training part and testing part which always contains only images from printer

Class Name	Only printer		Printer+Andorid	
	Train	Test	Train	Test
Button	9658	2426	16242	2426
StaticText	4626	1234	15197	1234
EditText	343	87	1425	87
StaticImage	984	262	3386	262
CheckBox	279	59	1329	59
Switch	49	15	473	15
RadioButton	48	23	1014	23

Tab. 5.2: Distribution of classes in datasets

dataset. Because of the insufficient dataset size and underrepresented classes, I have decided not to use the validation dataset that is commonly used in similar scenarios. See the class distribution in Tab. 5.2.

5.2.2 Initial Training

In the first experiment, the *ssd_inception_v2* model was trained with the same parameters that were used in the previous stage, but longer. After 24k steps on the printer dataset, the network achieved the $mAP = 0.598$. We can see that the evaluation loss in figure 5.1 dropped very fast within the first 8k steps but then it has a very small decreasing trend which might be caused by a small learning rate which was set to 0.004.

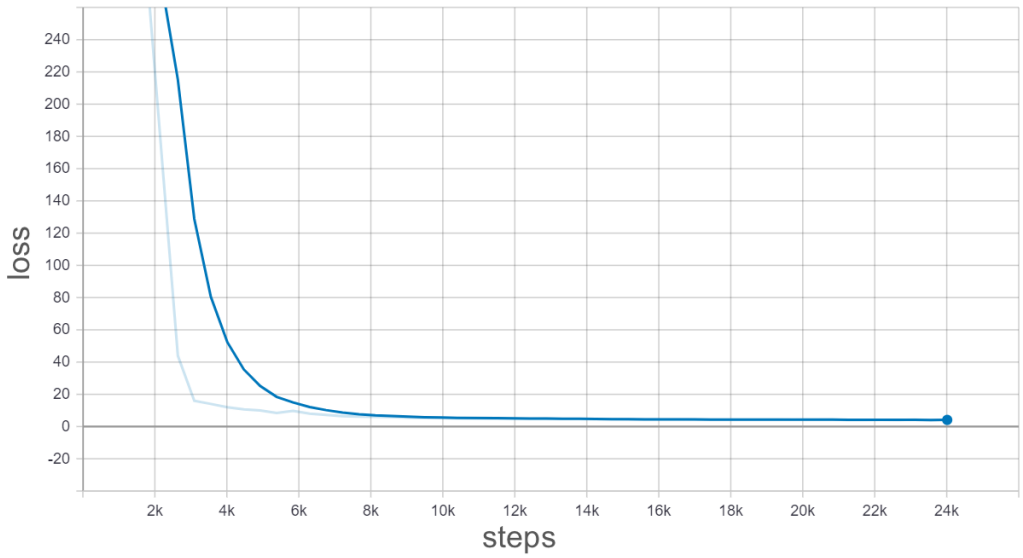
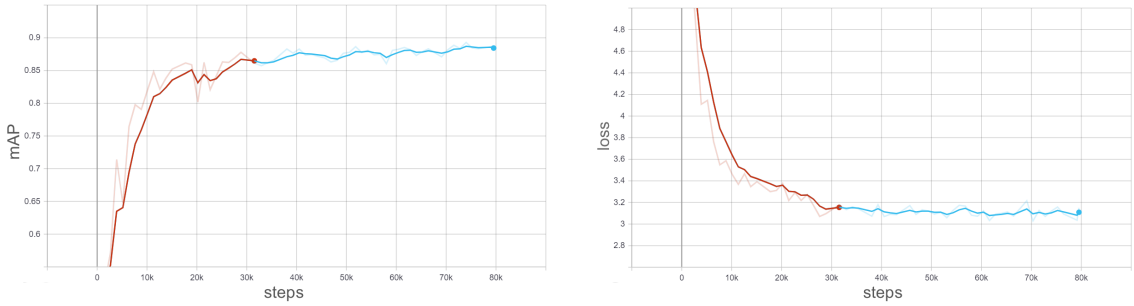


Fig. 5.1: Evaluation Loss of the First Experiment

In the second experiment, I have reduced the input image size to $300x300px$ to make the training faster, and because it was the original input size of the pre-trained network. I also increased the learning rate to 0.009 and added exponential decay. This resulted in a drastic improvement of the model performance. After 24k steps (the same number of steps as in the previous experiment) the model achieved $mAP = 0.842$ with a higher decreasing trend of evaluation loss. After 80k steps the model further improved the precision to $mAP = 0.883$ (see Fig. 5.2a and Fig. 5.2b). When continuing to train for more than 80k steps the model started to over-train. I also tried to train the model with increased input image size for 80k steps which slightly improved the precision to $mAP = 0.904$.



(a) mAP (b) Evaluation Loss

Fig. 5.2: The Second Experiment

5.2.3 Effect of Different Datasets

In the next series of experiments, I explored the effect of all three dataset variations on model precision. To make sure that the conclusion of this experiment applies to all models, I trained one model from each architecture. On top of that, from *Faster R-CNN* models was chosen the most precise model using the Resnet50 (see Tab. 5.4) and from *SSD* was chosen the fastest model using Mobilenet v1 (see Tab. 5.3) so the selected models are as diverse as possible.

We can see that the overall mAP is very similar for all datasets variations. In the case of *SSD* architecture it is around 0.6 and in the case of *Faster R-CNN*, it is around 0.9. This result is interesting, especially the comparison between the denoised and the raw Printer+Android datasets. Even though the model trained in the denoised dataset is able to learn much faster, the resulting precision is very similar. This indicates, that it may not be worth doing the image preprocessing, especially in my case when the current implementation of image denoising is very slow.

Category	Printer dat.		Mixed dat.		Mixed denoised dat.	
	Precision	Recall	Precision	Recall	Precision	Recall
StaticText	0.88	0.74	0.87	0.72	0.88	0.74
EditText	0.88	0.80	0.84	0.75	0.88	0.80
RadioButton	1.00	0.35	0.59	0.96	0.81	0.96
Switch	0.80	0.27	0.15	0.13	0.78	0.47
CheckBox	0.90	0.80	0.97	0.66	0.83	0.76
Button	0.91	0.91	0.87	0.90	0.91	0.89
StaticImage	0.74	0.68	0.68	0.68	0.80	0.72
STEPS	13k		31k		13k	
$mAP^{IoU=0.50}$	0.58		0.59		0.60	

Tab. 5.3: SSD Mobilenet v1 PPN Trained on three Datasets ("Mixed dat." is a dataset that consists of both Printer and Android images)

Category	Printer dat.		Mixed dat.		Mixed denoised dat.	
	Precision	Recall	Precision	Recall	Precision	Recall
StaticText	0.95	0.90	0.95	0.87	0.94	0.88
EditText	0.89	0.82	0.87	0.79	0.89	0.77
RadioButton	1.00	0.96	1.00	1.00	1.00	0.91
Switch	1.00	1.00	1.00	1.00	1.00	1.00
CheckBox	0.92	0.93	0.93	0.93	1.00	0.85
Button	0.96	0.97	0.95	0.96	0.96	0.95
StaticImage	0.85	0.85	0.87	0.81	0.84	0.85
STEPS	31k		58k		13.5k	
$mAP^{IoU=0.50}$	0.90		0.91		0.89	

Tab. 5.4: Faster R-CNN Resnet50 Trained on three Datasets ("Mixed dat." is a dataset that consists of both Printer and Android images)

	CPU Speed [ms]	GPU Speed [ms]
SSD Inception v2	297	42
Faster R-CNN Resnet50	3066	89
SSD Mobilenet_v1 PPN	222	26
SSD Lite Mobilenet_v2	74	27

Tab. 5.5: Inference Speed Comparison

There is also a question, whether it is better to use only the printer dataset alone or the combination of printer and Android dataset because the precision of trained models on both datasets is very similar (see Tab. 5.3 and Tab. 5.4). In general, the larger and more diverse is the dataset, the better and more general model is learned. In the case of printer dataset, adding the Android images also tackles the problem with too few examples of Switches and Checkboxes in the dataset. This implies, that it might be a good idea to do the model optimization on only printer datasets to speed up the training and then train the final model on the combination of both datasets in order to train a more robust model.

5.2.4 Detection Speed

The last series of experiments examine the detection speed of the selected models. Since the final model will be deployed on the machine without GPU, I have decided to measure the speed of the models on a computer with the following specifications:

- Processor: Inter Core i5-4310U CPU @ 2.00Ghz,
- RAM: 16GB,
- System: 64-bit Windows 10.

This test was conducted by loading a frozen inference graph and measuring how long it takes to infer one image. That was repeated for 100 images and the result was averaged. In table 5.5, you can see also comparison with the speed of the models on GPU (Nvidia GeForce GTX TITAN X card) which was taken from the TensorFlow model zoo documentation [60].

As you can see in the table 5.5, the *R-CNN* is surprisingly much more slower than expected. Especially when we compare the relative difference between *Faster R-CNN* and *SSD Inception_v2* measured on GPU and CPU. In the case of GPU, *Faster R-CNN* is 2 times slower but when we compare the same architecture speeds on CPU, the *R-CNN* is 10 times slower. These results indicate that models using the *R-CNN* architecture might not be applicable for my task due to a very slow speed.

5.3 Results Discussion

In the previous chapter, four selected models were trained using various training parameters. This chapter will discuss performance of fully trained models using the knowledge gained from the previous chapter and summarizes their suitability for a given task.

In the table 5.6, we can see the selected models, fully trained on combination of printer and Android dataset without denoising. This decision was conducted based on the data in the section 5.2.3 which did not show any significant improvement of model performance with denoised input images. Moreover, the denoising in its current implementation would drastically slow down the whole detection process (see the effect of denoising in Tab. 5.3 and Tab. 5.4).

The first observation from the Tab. 5.6 is that the *Faster R-CNN* architecture is not much more accurate than *SSD Inception v2* which is an architecture focusing rather on speed than accuracy. We can see that the *R-CNN* shows better results for higher threshold levels of IoU which means that it is able to locate the objects more accurately. However, the input image size of *R-CNN* network is larger which definitely affect the detection accuracy. Besides that, the detection speed of the *R-CNN* is approximately 10 times worse than *SSD Inception v2* which is a potential hurdle because one of the task requirements is the worst-case detection speed around one second. I also measured the average detection speed of *Faster R-CNN Resnet50* with input image size *300x300px* which resulted in average speed 2466 ms and *Faster R-CNN Inception v2* with input image size *600x960px* which resulted in average detection speed 1178ms. These times are better but still most likely too slow.

Considering the speed, I have selected the *SSD Mobilenet v1* and *SSD Lite Mobilenet v2* solely based on the detection speed. The results of training on COCO dataset imply, that *SSD Mobilenet v1* should be slightly faster with worse *mAP* (see the Tab.5.5 and 5.1). Surprisingly, the overall performance of *SSD Lite* on the printer dataset is much better (see Tab. 5.6). The $AP^{IoU=0.50}$ of *SSD Lite* is higher by 14% and it is three times faster. Since the input image size of *SSD Mobilenet v1* is two time bigger than the input of *SSD Lite*, I also trained the *SSD Mobilenet v1* on the same input image size. That reduced the detection speed of *SSD Mobilenet v1* to 71ms but also decreased already much worse model accuracy.

$Metric$	$SSD_{Inceptionv2}$	$RCNN_{Resnet50}$	$SSD_{Mobilenetv1}$	SSD_{Lite}
$AP^{IoU=0.50:0.95}_{area=all}$	0.542	0.602	0.360	0.533
$AP^{IoU=0.50}_{area=all}$	0.904	0.904	0.687	0.831
$AP^{IoU=0.75}_{area=all}$	0.602	0.675	0.336	0.608
$AP^{IoU=0.50:0.95}_{area=small}$	0.219	0.440	0.119	0.14
$AP^{IoU=0.50:0.95}_{area=medium}$	0.568	0.597	0.335	0.544
$AP^{IoU=0.50:0.95}_{area=large}$	0.607	0.644	0.542	0.644
$AR^{IoU=0.50:0.95}_{area=all}$	0.296	0.322	0.238	0.315
$AR^{IoU=0.50:0.95}_{area=all}$	0.569	0.595	0.425	0.546
$AR^{IoU=0.50:0.95}_{area=all}$	0.642	0.660	0.485	0.601
$AR^{IoU=0.50:0.95}_{area=small}$	0.294	0.486	0.159	0.172
$AR^{IoU=0.50:0.95}_{area=medium}$	0.654	0.656	0.476	0.612
$AR^{IoU=0.50:0.95}_{area=large}$	0.715	0.708	0.641	0.709
CPU Speed [ms]	297	3066	222	74
Img. Size [px]	640x360	1024x576	600x600	300x300

Tab. 5.6: Fully Trained Models on Printer+Andorid dataset

Category	$Precision^{IoU=0.5}$	$Recall^{IoU=0.5}$
StaticText	0.97	0.61
EditText	0.97	0.44
RadioButton	1.00	0.91
Switch	1.00	0.73
CheckBox	1.00	0.88
Button	0.97	0.87
StaticImage	0.91	0.79

Tab. 5.7: Per Class Precision and Recall of *SSD Inception v2*

When we take into account all previously mentioned, the networks using the *SSD* architecture seem to be a better fit for my task due to the better detection speed. From the four fully trained architectures, the *SSD Inception v2* achieves precision, that is even comparable to the *Faster R-CNN*. The detailed per class precision and recall can be seen in the table 5.7. You can also see the example of detected UI elements in the figure 5.3 where on the left side are detected boundary boxes and on the right side are ground-truth boxes.

The trained models are saved in folder *code/assets/trained_models*, so it is pos-

sible to try them on a small subset of test images in folder *code/assets/images* by running either *code/detect_single.py* on single image from Powershell Prompt:

```
python detect_single.py
  -model= <path to frozen_inference_graph.pb>
  -img= <path to img.png>
  -labelmap= <path to labelmap.pbtxt>
```

or *code/detect_from_folder.py* for all images in folder:

```
python detect_from_folder.py
  -model= <path to frozen_inference_graph.pb>
  -img_dir= <path to img folder>
  -labelmap= <path to labelmap.pbtxt>
```

The labelmap file can be found in the folder *assets/labelmap_7_classes.pbtxt*. To run the code, it is necessary to install all the dependencies. The easiest way to do it is by using Anaconda package management system [66]. The copy of my virtual environment can be created by using the exported configuration file located at *code/assets/environment.yml*:

```
conda env create -f environment.yml
```

Afterwards you can activate the environment:

```
conda activate tf1_cpu
```

and run the commands for detection above. This environment is using the CPU version of Tensorflow so it should work without any need to deal with CUDA and cuDNN.

Fig. 5.3: Detection Outputs with Confidence Scores. The image on the left shows detected boxes and image on right shows the ground-truth boxes

6 Conclusion

This thesis deals with the detection of UI elements in a photo of a screen using machine learning techniques, namely using convolutional neural networks. This work is a part of a system that is used for automatic, black-box testing of printers using a robotic arm with a stylus. The goal of this work is to further automate the creation of test cases. In the current version of the system, user has to draw boundary boxes around the UI elements by hand which is a slow tedious job. Implementation of this thesis into the system will ease and speed up the test setup.

The first part of the thesis introduces basic concepts of neural networks, it's difference from convolutional neural networks, and the techniques that are commonly used and thus lays theoretical foundations for the rest of the work. The second chapter builds on the first chapter by reviewing scientific papers of CNN architectures. It briefly explains the obsolete architectures that paved the way for the modern architectures that are discussed in more detail. The second part of this chapter explains the principles of currently most used object detection meta-architectures R-CNN, SSD and YOLO.

Since the performance of machine learning algorithms is strongly dependent on datasets, the subsequent chapter discusses data preprocessing and analysis of the available printer dataset. Based on the analysis was created a list of seven classes into which we grouped the UI elements during the annotation of the printer dataset. As the available printer dataset contains only 1218 images, we discussed the possibilities of dataset extension by some publicly available dataset of UI screens. During the research were found two projects, RICO and ReDraw, that collected and published a dataset of Android UI screens. Since the ReDraw dataset was created with intention of UI element detection as well, I have decided to use part of this dataset to extend the printer dataset. Due to the inconsistency of the ReDraw dataset, it was necessary to convert the XML files of each Android screen into a PASCAL VOC format and use the CVAT annotation tool to correct the boundary boxes in the ReDraw dataset. The resulting dataset size has grown to 2 231 images.

The fourth chapter discusses the possible machine learning frameworks that could be used to train the object detection models. Based on the research was selected high level, task oriented, object detection framework called *Tensorflow Object Detection API*. The chapter explains how to set up the framework and how to tune the training parameters in order to train the pretrained object detection models from the model zoo. This chapter also presents some of the created scripts that were created to overcome some shortcoming of the framework such as periodic backup of trained model checkpoints and more detailed model evaluation.

The fifth chapter of this thesis is devoted to finding a best fitting model for

a given task. It combines the knowledge gained from the literature research and uses transfer-learning to train pretrained models of two selected object detection meta-architectures, *Faster R-CNN* and *SSD*, using te framework *Tensorflow Object Detection API*. The model training is divided into two logical parts. In the first part, we preselected 10 models out of 38 models available in the TFOD API and train them briefly on a small subset of the printer dataset. Based on the models performance on printer dataset and performance on COCO dataset from the TFOD documentation, four models were selected. These models were further trained in the second step where we conducted series of experiments whose goal was to find pseudo-optimal settings of training parameters and decide which model architecture suits best for the given task.

One of the experiments examined the effect of image denoising and demoiréing on the model accuracy. The results of this experiment showed, that the models trained on the preprocessed images are able to finish training faster, however with similar accuracy. We also tested the effect of dataset extension by the Android dataset. Even though the experiment did not show significant improvement of performance between model train only on printer dataset and model tarind on extended dataset, I have decided to use the extended dataset since the resulting accuracy is similar and the trained model should be more general since it is trained on more diverse data.

The results of training showed that due to the requirement on detection speed, it is clear that the *Faster R-CNN* architecture is not suitable because the deployed machine will not have GPU and speed tests performed on CPU showed average speed in the range from 1.1 second to 3 seconds. The most promising architecture is *SSD Inception v2* that was able to achieve the $mAP^{IoU=50}$ 0.904 which is the same as the *Faster R-CNN* while keeping the average detection speed 297ms. If we would prioritize the detection speed over accuracy we can consider using *SSD Lite Mobilenet v2* which is able to process images with average speed 74ms but at cost of $mAP^{IoU=50}$ that drops to 0.821.

Performance of the trained models achieves sufficient levels and it fulfills all requirements in order to be deployed and used in real-life applications. Nevertheless, there is still room for improvement. The biggest bottleneck of this project is limited dataset which makes it more challenging to properly train and test the models. There is a possibility to improve the dataset quality as there is an ongoing project whose goal is to automatically map all UI screens of the given printer. This way it is possible to automatically create a more comprehensive dataset that can be automatically annotated by one of the models trained in this thesis and afterward corrected by hand. Other than that, this assignment was fulfilled at all points.

Bibliography

[1] Evolution's 'big bang' explained (and it's slower than predicted), Mike Lee, University of Adelaide, Available at: https://theconversation.com/evolutions-big-bang-explained-and-its-slower-than-predicted-18098

[2] Rosenblat, F. "The perceptron: A probabilistic model for information storage and organization in the brain." Psychological Review 65.6 (1958): 386-408. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf

[3] Sebastian Raschka, Single-Layer Neural Networks and Gradient Descent, Available at: https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html

[4] Deeplearning - Overview of Convolution Neural Network, Available at: https://www.zybuluo.com/hongchenzimo/note/1086311

[5] Vladimir Stojov, Nikola Koteli,Petre Lameski, Eftim Zdravevski, Application of machine learning and time-series analysis for air pollution prediction, Available at: https://www.researchgate.net/publication/327765620_Application_of_machine_learning_and_time-series_analysis_for_air_pollution_prediction

[6] UNSW Sydney, Backpropagation, Available at: https://www.cse.unsw.edu.au/~cs9417ml/MLP2/BackPropagation.html

[7] Pawan Jain, Complete Guide of Activation Functions, Available at: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

[8] Kamil Krzyk, Coding Deep Learning for Beginners — Linear Regression (Part 2): Cost Function, Available at: https://towardsdatascience.com/coding-deep-learning-for-beginners-linear-regression-part-2-cost-function-49545303d29f

[9] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324., Available at: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

[10] Lindsay, Grace. "Convolutional Neural Networks as a Model of the Visual System: Past, Present, and Future." Journal of Cognitive Neuroscience (2020): 1-15. Available at: https://arxiv.org/pdf/2001.07092.pdf

[11] Wikipedia: Convolution. Available at: https://arxiv.org/pdf/2001.07092.pdf

[12] Stanford University. CS231n Convolutional Neural Networks for Visual Recognition, Available at: https://cs231n.github.io/convolutional-networks/

[13] Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806. Available at: https://arxiv.org/pdf/1412.6806.pdf

[14] Subir Varma and Sanjiv Das, Deep Learning, 2018-09-27, Available at: https://srdas.github.io/DLBook/

[15] Dewan Nautiyal, GeeksforGeeks, Underfitting and Overfitting in Machine Learning https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

[16] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." Available at: https://arxiv.org/pdf/1502.03167v3.pdf

[17] Adrian Rosebrock, Intersection over Union (IoU) for object detection, Pyimagesearch, Available at: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

[18] Manal El Aidouni, Evaluating Object Detection Models: Guide to Performance Metrics, Available at: https://manalelaidouni.github.io/manalelaidouni.github.io/Evaluating-Object-Detection-Models-Guide-to-Performance-Metrics.html

[19] Raimi Karim, Illustrated: 10 CNN Architectures, Available at: https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d

[20] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In Advances in neural information processing systems, pp. 1097-1105. 2012. Available at: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[21] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014), Available at: https://arxiv.org/pdf/1409.1556.pdf

[22] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1-9. 2015., Available at: https://arxiv.org/abs/1409.4842

[23] Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013), Available at: https://arxiv.org/abs/1312.4400

[24] Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2818-2826. 2016., Available at: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Szegedy_Rethinking_the_Inception_CVPR_2016_paper.html

[25] Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber. "Highway networks." arXiv preprint arXiv:1505.00387 (2015)., Available at: https://arxiv.org/pdf/1505.00387.pdf

[26] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016., Available at: http://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

[27] Liu, Li, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. "Deep learning for generic object detection: A survey." International journal of computer vision 128, no. 2 (2020): 261-318., Available at: https://link.springer.com/article/10.1007/s11263-019-01247-4

[28] Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 580-587. 2014., Available at: http://openaccess.thecvf.com/content_cvpr_2014/papers/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.pdf

[29] Uijlings, Jasper RR, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. "Selective search for object recognition." International journal of computer vision 104, no. 2 (2013): 154-171., Available at: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf

[30] Joyce Xu, Deep Learning for Object Detection: A Comprehensive Review, Sep 11, 2017, Available at: https://towardsdatascience.com/deep-learning-for-object-detection-a-comprehensive-review-73930816d8d9

[31] Girshick, Ross. "Fast r-cnn." In Proceedings of the IEEE international conference on computer vision, pp. 1440-1448. 2015., Available at: https://arxiv.org/pdf/1504.08083.pdf

[32] Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks." In Advances in neural information processing systems, pp. 91-99. 2015., Available at: https://arxiv.org/pdf/1506.01497.pdf

[33] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. "Ssd: Single shot multibox detector." In European conference on computer vision, pp. 21-37. Springer, Cham, 2016., Available at: https://arxiv.org/pdf/1512.02325.pdf

[34] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 779-788. 2016., Available at: https://arxiv.org/abs/1506.02640

[35] COCO, Common Objects in Context, Evaluation metrics, Available at: http://cocodataset.org/#detection-eval

[36] Glenn Jocher, EPRECISION-RECALL CURVE, Available at: https://github.com/ultralytics/yolov3/issues/898

[37] Fukushima, Kunihiko. "Neocognitron: A hierarchical neural network capable of visual pattern recognition." Neural networks 1, no. 2 (1988): 119-130., Available at: http://vision.stanford.edu/teaching/cs131_fall1415/lectures/Fukushima1988.pdf

[38] Yann LeCun, Corinna Cortes, Christopher J.C. Burges. (n.d.). THE MNIST DATABASE. Retrieved April 6, 2020, from http://yann.lecun.com/exdb/mnist/

[39] Alex Krizhevsky. (n.d.). CIFAR Datatset. Retrieved April 6, 2020, from https://www.cs.toronto.edu/~kriz/cifar.html

[40] CIFAR-10: Object recognition, Leaderboard: https://www.kaggle.com/c/cifar-10/leaderboard

[41] CIFAR-100: Object recognition, Leaderboard: https://www.kaggle.com/c/ml2016-7-cifar-100/leaderboard

[42] PASCALVOC: Official website. Available at: http://host.robots.ox.ac.uk/pascal/VOC/

[43] Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L. (2014, September). Microsoft coco: Common objects in context. In European conference on computer vision (pp. 740-755). Springer, Cham. Available: https://arxiv.org/abs/1405.0312

[44] COCO: Common objects in context, Official website, Available: http://cocodataset.org/#home

[45] Boaz Shmueli, Multi-Class Metrics Made Simple, Part I: Precision and Recall, Available: https://towardsdatascience.com/multi-class-metrics-made-simple-part-i-precision-and-recall-9250280bddc2

[46] Rico: A Mobile App Dataset for Building Data-Driven Design Applications, UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN, Available at: http://interactionmining.org/rico

[47] The ReDraw Dataset: A Set of Android Screenshots, GUI Metadata, and Labeled Images of GUI Components, Moran, Kevin; Bernal-Cardenas, Carlos; Curcio, Michael; Bonett, Richard; Poshyvanyk, Denys, Available at: https://zenodo.org/record/2530277#.Xoy8SNP7TOQ

[48] Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps, Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, Denys Poshyvanyk, Avauilable at: https://arxiv.org/abs/1802.02312

[49] LabelImg - graphical image annotation tool. Available at: https://github.com/tzutalin/labelImg

[50] VIA - VGG Image Annotator, Abhishek Dutta, Ankush Gupta and Andrew Zisserman, Department of Engineering Science, University of Oxford. Available at: http://www.robots.ox.ac.uk/~vgg/software/via/

[51] Labelbox, Available at: https://labelbox.com/

[52] Computer Vision Annotation Tool, CVAT, developed by Intel, Available at: https://github.com/opencv/cvat

[53] OpenCV documentation, Image Denoising, Available at: https://docs.opencv.org/3.4/d5/d69/tutorial_py_non_local_means.html

[54] Docker, Solomon Hykes, Available at: https://www.docker.com/

[55] AutoML for Data Augmentation Barış Özmen, Available at: https://blog.insightdatascience.com/automl-for-data-augmentation-e87cf692c366

[56] Buades, Antoni, Bartomeu Coll, and Jean-Michel Morel. "Non-local means denoising." Image Processing On Line 1 (2011): 208-212., Available at: https://www.ipol.im/pub/art/2011/bcm_nlm/article.pdf

[57] Cesar De la Torre, Principal Program Manager, .NET, Training Image Classification/Recognition models based on Deep Learning & Transfer Learning with ML.NET Available at: https://devblogs.microsoft.com/cesardelatorre/training-image-classification-recognition-models-based-on-deep-learning-transfer-learning-with-ml-net/

[58] Keras Documentation, Models for image classification with weights trained on ImageNet, Available at: https://keras.io/applications/

[59] Tensorflow Object Detection API, Available at: https://github.com/tensorflow/models/tree/master/research/object_detection

[60] Tensorflow Object Detection API, Tensorflow detection model zoo, Available at: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

[61] Nvidia DEVELOPER, Recommended GPU for Developers, Available at: https://developer.nvidia.com/cuda-gpus

[62] Tensorflow documentation, Build from source on Windows, Available at: https://www.tensorflow.org/install/source_windows#tested_build_configurations

[63] COCO API, Available at: https://github.com/cocodataset/cocoapi

[64] COCO API, Unofficial Windows version, Available at: https://github.com/philferriere/cocoapi

[65] Santiago Valdarrama, Confusion Matrix in Object Detection with TensorFlow, Available at: https://github.com/svpino/tf_object_detection_cm

[66] Conda documentation, Available at: https://docs.conda.io/projects/conda/en/latest/index.html

List of symbols, quantities and abbreviations

UI	User Interface
KNS	Konvoluční neuronová síť
MSE	Mean Square Error
BP	Backpropagation
CNN	Convolutional neural network
ConvNet	Convolutional neural network
RNP	Region Proposal Network
SSD	Single-Shot Detector
YOLO	You Only Look Once
mAP	mean Average Precision
IoU	Intersection over Union
MNIST	The Modified National Institute of Standards and Technology
OCR	Optical character recognition
KNN	K-Nearest Neighbors
CVAT	Computer Vision Annotation Tool
DFT	Discrete Fourier Transform
TFOD	Tensorflow Object Detection API

List of appendices

A Configuration file `pipeline.config`

```
model {
  ssd {
    num_classes: 7
    image_resizer {
      fixed_shape_resizer {
        height: 300
        width: 300
      }
    }
    feature_extractor {
      type: "ssd_mobilenet_v1_ppn"
      conv_hyperparams {
        regularizer {
          l2_regularizer {
            weight: 3.99999989895e-05
          }
        }
        initializer {
          random_normal_initializer {
            mean: 0.0
            stddev: 0.00999999977648
          }
        }
        activation: RELU_6
        batch_norm {
          decay: 0.97000002861
          center: true
          scale: true
          epsilon: 0.0010000000475
        }
      }
      override_base_feature_extractor_hyperparams: true
    }
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
```

```
    }
  }
  matcher {
    argmax_matcher {
      matched_threshold: 0.5
      unmatched_threshold: 0.5
      ignore_thresholds: false
      negatives_lower_than_unmatched: true
      force_match_for_each_row: true
      use_matmul_gather: true
    }
  }
  similarity_calculator {
    iou_similarity {
    }
  }
  box_predictor {
    weight_shared_convolutional_box_predictor {
      conv_hyperparams {
        regularizer {
          l2_regularizer {
            weight: 3.99999989895e-05
          }
        }
        initializer {
          random_normal_initializer {
            mean: 0.0
            stddev: 0.00999999977648
          }
        }
        activation: RELU_6
        batch_norm {
          decay: 0.97000002861
          center: true
          scale: true
          epsilon: 0.0010000000475
          train: true
        }
      }
      depth: 512
      num_layers_before_predictor: 1
```

```
81          kernel_size: 1
82          class_prediction_bias_init: -4.59999990463
83          share_prediction_tower: true
84        }
85      }
86      anchor_generator {
87        ssd_anchor_generator {
88          num_layers: 6
89          min_scale: 0.15000000596
90          max_scale: 0.949999988079
91          aspect_ratios: 1.0
92          aspect_ratios: 2.0
93          aspect_ratios: 0.5
94          aspect_ratios: 3.0
95          aspect_ratios: 0.333299994469
96          reduce_boxes_in_lowest_layer: false
97        }
98      }
99      post_processing {
100       batch_non_max_suppression {
101         score_threshold: 0.300000011921
102         iou_threshold: 0.600000023842
103         max_detections_per_class: 100
104         max_total_detections: 100
105       }
106       score_converter: SIGMOID
107     }
108     normalize_loss_by_num_matches: true
109     loss {
110       localization_loss {
111         weighted_smooth_l1 {
112         }
113       }
114       classification_loss {
115         weighted_sigmoid_focal {
116           gamma: 2.0
117           alpha: 0.75
118         }
119       }
120       classification_weight: 1.0
121       localization_weight: 1.5
```

```
122     }
123     encode_background_as_zeros: true
124     normalize_loc_loss_by_codesize: true
125     inplace_batchnorm_update: true
126     freeze_batchnorm: false
127   }
128 }
129 train_config {
130   batch_size: 4
131   data_augmentation_options {
132     random_horizontal_flip {
133     }
134   }
135   sync_replicas: true
136   optimizer {
137     momentum_optimizer {
138       learning_rate {
139         cosine_decay_learning_rate {
140           learning_rate_base: 0.699999988079
141           total_steps: 50000
142           warmup_learning_rate: 0.13330000639
143           warmup_steps: 2000
144         }
145       }
146       momentum_optimizer_value: 0.899999976158
147     }
148     use_moving_average: false
149   }
150   fine_tune_checkpoint: "<path-to-model.ckpt>"
151   num_steps: 50000
152   startup_delay_steps: 0.0
153   replicas_to_aggregate: 8
154   max_number_of_boxes: 100
155   unpad_groundtruth_tensors: false
156 }
157 train_input_reader {
158   label_map_path: "<path-to-labelmap.pbtxt>"
159   tf_record_input_reader {
160     input_path: "<path-to-train.record>"
161   }
162 }
```

```
eval_config {
  num_examples: 244
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
  eval_interval_secs: 600
}
eval_input_reader {
  label_map_path: "<path-to-labelmap.pbtxt>"
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "<path-to-test.record>"
  }
}
```

B PASCAL VOC XML file

```
<annotation>
 <filename>2.png</filename>

 <source>
   <database>Unknown</database>
   <annotation>Unknown</annotation>
   <image>Unknown</image>
 </source>

 <size>
   <width>1280</width>
   <height>724</height>
   <depth>3</depth>
 </size>

 <segmented>0</segmented>

 <object>
   <name>StaticText</name>
   <bndbox>
     <xmin>339.8642578125</xmin>
     <ymin>443.3271484375</ymin>
     <xmax>933.9910888671875</xmax>
     <ymax>498.15000915527344</ymax>
   </bndbox>
 </object>
   <object>
   <name>Button</name>
   <bndbox>
     <xmin>506.755859375</xmin>
     <ymin>629.916015625</ymin>
     <xmax>772.2005004882812</xmax>
     <ymax>715.5926666259766</ymax>
   </bndbox>
 </object>
</annotation>
```

C Media content

- `/` — root folder
  - `xhorak59.pdf` — Thesis document
  - `dataset` — Small example of dataset
    - `android`
      - `images`
      - `annotations`
    - `printer`
      - `images`
      - `annotations`
  - `code` — Used code
    - `detect_from_folder.py` — Run model on all images in folder
    - `detect_single.py` — Run model on single image
    - `detector.py` — Detector class
    - `evaluate.py` — The model evaluation script
    - `README.md`
    - `.gitignore`
    - `assets`
      - `images` — Subset of test images
      - `trained_models` — Trained models
      - `environment.yml` — Exported environment configuration
      - `labelmap_7_classes.pbtxt` — Labelmap
    - `dataset_objects` — Objects for parser
    - `models` — Tensorflow Object detection library files
    - `utilities` — Utility scripts